2008-03

Implementation of a cyclostationary spectral analysis algorithm on an SRC reconfigurable computer for real-time signal processing

Upperman, Gary J.

Monterey California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

IMPLEMENTATION OF A CYCLOSTATIONARY SPECTRAL ANALYSIS ALGORITHM ON AN SRC RECONFIGURABLE COMPUTER FOR REAL-TIME SIGNAL PROCESSING

by

Gary J. Upperman

March 2008

Thesis Advisor:	Douglas J. Fouts
Co-Advisor:	Phillip E. Pace

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** March 2008	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE Implementation of a Cyclostationary Spectral Analysis Algorithm on an SRC Reconfigurable Computer for Real-Time Signal Processing			**5. FUNDING NUMBERS**
6. AUTHOR(S) Gary J. Upperman			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Center for Joint Service Electronic Warfare Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Security Agency, Fort Mead, MD; Office of Naval Research, Arlington, VA			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

This thesis describes a near-real-time method of detecting low probability of intercept (LPI) emissions. A cyclostationary spectral analysis algorithm developed by the Center for Joint Services Electronic Warfare at the Naval Postgraduate School was implemented on the SRC-6 reconfigurable computer. This thesis is part of a larger project investigating the use of the SRC-6 for electronic intelligence detection and processing. Cyclostationary processing transforms a received signal into a frequency-cycle frequency domain which can have detection advantages over a time-frequency domain transformation. When performed at near-real-time processing speed, the algorithm can be used to detect and classify LPI emissions. The performance of the algorithm on the SRC-6 is compared to equivalent implementations in MATLAB and the C programming language.

14. SUBJECT TERMS Low Probability of Intercept, LPI, Cyclostationary Processing, Reconfigurable Computing, SRC, SRC-6, Electronic Intelligence, ELINT			**15. NUMBER OF PAGES** 124 **16. PRICE CODE**
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

IMPLEMENTATION OF A CYCLOSTATIONARY SPECTRAL ANALYSIS ALGORITHM ON AN SRC RECONFIGURABLE COMPUTER FOR REAL-TIME SIGNAL PROCESSING

Gary J. Upperman
Civilian, Department of Defense
B.S., Valparaiso University, 2001

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
March 2008

Author: Gary J. Upperman

Approved by: Douglas J. Fouts
Thesis Advisor

Phillip E. Pace
Co-Advisor

Jeffrey B. Knorr
Chairman, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis describes a near-real-time method of detecting low probability of intercept (LPI) emissions. A cyclostationary spectral analysis algorithm developed by the Center for Joint Services Electronic Warfare at the Naval Postgraduate School was implemented on the SRC-6 reconfigurable computer. This thesis is part of a larger project investigating the use of the SRC-6 for electronic intelligence detection and processing. Cyclostationary processing transforms a received signal into a frequency-cycle frequency domain which can have detection advantages over a time-frequency domain transformation. When performed at near-real-time processing speed, the algorithm can be used to detect and classify LPI emissions. The performance of the algorithm on the SRC-6 is compared to equivalent implementations in MATLAB and the C programming language.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF SYMBOLS, ACRONYMS, AND ABBREVIATIONS

ADC	Analog-to-Digital Converter
A.K.A	Also Known As
df	Frequency Resolution (128 Hz for this thesis)
ELINT	Electronic Intelligence
FAM	FFT Accumulation Method
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
fs	Sampling Frequency
FSK	Frequency Shift Keying
GPIO	General Purpose Input Output
I-channel	In-phase Channel
LPI	Low Probability of Interception
M	Grenander's Uncertainty Condition (2 for this thesis)
MAP	Multi-Adaptive Processing® Board
OBM	On Board Memory
PSK	Phase Shift Keying
Q-channel	Quadrature Channel
VHDL	Very High Speed Integrated Circuit Hardware Description Language

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank David Caliga of SRC Computers, Inc. Mr. Caliga provided valuable information on the SRC-6 computer at the Naval Postgraduate School. He was quick to respond to questions, and he also provided the Naval Postgraduate School with the fast Fourier transform algorithms used in this thesis.

Professors Douglas Fouts and Phillip Pace of the Naval Postgraduate School were co-advisors and provided valuable information about the Naval Postgraduate School's SRC-6 computer. Professor Pace also provided the initial algorithm which served as a starting point for this research effort.

Dan Zulaica at the Naval Postgraduate School provided some key information about the school's SRC-6. He also maintained the computer system and installed the fast Fourier transform algorithms the school received from SRC Computers.

Many thanks are given to our research sponsors. Financial support was provided by the National Security Agency and the Office of Naval Research (code 312, Arlington, VA). Without their interest in the larger scope of the project, this research could not have been completed.

My wife, Teresa, provided the never ending support that I needed from home. I am sure that the *sacrifices* she made to relocate from the 'high desert' to Monterey can never be repaid! I appreciate the patience of my daughter, Samantha, while I spent long hours away from home, and to her grandpa Ken for his willingness to keep an eye on her during those long hours.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

The Naval Postgraduate School Center for Joint Electronic Warfare conducts research into the field of low probability of interception (LPI) signals and intercept signal processing. The focus of this thesis was to explore the feasibility of implementing a cyclostationary spectral analysis algorithm on an SRC-6 reconfigurable computer. This thesis is part of a larger project exploring and evaluating the use of the SRC-6 reconfigurable computer for processing electronic intelligence (ELINT) data. The collective objective of this effort is to use the SRC-6 computer as a real-time ELINT detection system designed to detect and classify LPI radar signals.

Recent developments in radar technology include the employment of LPI capabilities which are designed to make the task of intercepting radar signals more difficult. Using complex algorithms to process received LPI signals can result in threat detection in situations where a conventional intercept receiver would interpret the signal as noise or not notice the signal at all. Cyclostationary spectral analysis is a signal processing technique that models an input signal as a cyclic process and transforms that signal into a frequency-cycle-frequency domain. This transformation can be used to extract signal properties and result in detection of an LPI signal. A drawback to the complex algorithms used is the processing power and time required to obtain the results. Reconfigurable computers like the SRC-6 have been proposed as a solution to process these signals in near-real-time. The envisioned ELINT detection system would process the same signal using three different algorithms, compare the results, and deliver a unified decision on the modulation technique and other signal parameters. That information could then be used to identify the source of the signal and suggest possible countermeasure options.

This thesis started with a cyclostationary implementation in MATLAB developed in [3]. There are several different cyclostationary algorithms available and this thesis specifically focuses on the fast Fourier transform accumulation method which is highly efficient. The MATLAB algorithm was converted into standard C, tested, and finally

converted into an SRC-specific algorithm. The output and timing performance of all three algorithms were compared to determine the feasibility of using the SRC-6 for the cyclostationary spectral analysis processing. Test LPI signals were generated by the LPI Toolbox [3] and used in substitution of actual analog-to-digital converter samples.

The timing results showed that all three implementations produced results in less than one second. Although the implementation in standard C was comparable to MATLAB, they both outperformed the implementation on the SRC-6. Plots of the output showed that the MATLAB and C implementations were virtually identical. The output from the SRC implementation differed slightly from the other two, but resulted in the same overall conclusions about the LPI test signal parameters.

The cyclostationary spectral analysis implementation on the SRC-6 computer produced quality results in a timely manner. Therefore, it is feasible to use the SRC-6 computer to employ a cyclostationary spectral analysis algorithm as a part of an ELINT detection system. The results of the cyclostationary algorithm should be compared to the results of other processing algorithms (see [1] and [2]) in order to produce a quality, unified decision about the intercepted LPI signal.

I. INTRODUCTION

A. PURPOSE

Prior information and intelligence about a theater environment can improve the success of any military operation into that environment. Electronic intelligence (ELINT) is a key element of that information. In order to make the most of any information gathered, the data must be as accurate and current as possible. The capability to digitally process theater environment ELINT signals in real-time offers an advantage over an adversary on the battlefield.

Recent developments in radar technology include the employment of low probability of interception (LPI) and low probability of detection capabilities which are designed to make the task of intercepting radar signals more difficult. Using a complex algorithm, such as cyclostationary spectral analysis, to process these signals can result in threat detection and extraction of signal properties. A drawback to such a complex algorithm is the processing power and time required to obtain the results. Reconfigurable computers, like the SRC-6, have been proposed as a solution to process these signals in near-real-time. The result of such a system is the capability to carry it shipboard or onboard an aircraft or other vehicle where it can be used on the battlefield during an engagement.

B. OBJECTIVE

This thesis is part of a larger project exploring and evaluating the use of the SRC-6 reconfigurable computer for ELINT processing. The collective objective of this effort is to use the SRC-6 computer as an ELINT detection system whereby an LPI signal can be processed using multiple algorithms to produce a unified set of signal parameters. More details on the broader scope of the project are given in Chapter II. The feasibility of implementing a cyclostationary spectral analysis algorithm on the SRC-6 is the focus of this thesis and is explored in detail. The performance of the algorithm on the SRC-6 is

compared to equivalent implementations in C and MATLAB**.** Conclusions and recommendations are presented based on the results of this effort.

C. RELATED WORK

This thesis is part of a larger project. Several theses have been completed exploring the use of the SRC-6 as an ELINT detection system. A quadrature mirror filter bank, another LPI detection method, was completed in [1]. The Choi-Williams distribution in [2] was another detection algorithm explored for use on the SRC-6. More details on how these two detection schemes and the cyclostationary spectral analysis will fit together are given in Chapter II.

Professor Phillip Pace of the Naval Postgraduate School and his students have performed much of the research behind these algorithms in the Center for Joint Service Electronic Warfare. The three algorithms mentioned in the previous paragraph were implemented using MATLAB on a common personal computer prior to the implementations on the SRC-6. The cyclostationary algorithm developed in MATLAB is the basis and starting point for this thesis. Benchmarking of the SRC-6 at the Naval Postgraduate School was performed in [4] and [5], and has been built upon in subsequent research projects [6], [7], [8], and [9].

D. THESIS ORGANIZATION

The remainder of this thesis is organized as follows:

- Chapter II provides background on the ELINT detection system, the cyclostationary algorithm, and the SRC-6 reconfigurable computer.
- Chapter III discusses the cyclostationary algorithm, the translation from the MATLAB implementation to the C programming language and compares results.
- Chapter IV details the transition to the SRC-specific algorithm. Plots are used to show equivalency between the original MATLAB implementation and the final SRC-specific implementation.
- Chapter V presents the performance analysis of all three implementations.
- Chapter VI concludes with a summary of the results and suggestions for future work.

II. BACKGROUND

A. ELECTRONIC INTELLIGENCE (ELINT) DETECTION SYSTEM

The purpose of the collaborative effort of which this thesis is a part is to create an ELINT detection system capable of detecting LPI signals in near-real-time. Figure 1 shows a graphical depiction of the system as a whole. When the radar signal comes in from the receiver and any front-end processing (analog-to-digital conversion), it is sent to multiple channels of processing (Figure 1 currently shows three channels). The channels individually process the same signal using either time-frequency or bi-frequency techniques. The results of each channel are sent to a decision logic module which determines the type of waveform (LPI or otherwise) the radar is using. A detailed discussion and analysis of different LPI waveforms is given in [3]. The decision logic module sends results to the parameter extraction module which has already received the signal parameters from the detection algorithms. This information can now be used in conjunction with a previously-loaded mission data file (emitter look-up table) to determine which radar emitted the signal. This thesis focuses on one of the preprocessing decision algorithms, cyclostationary spectral analysis.

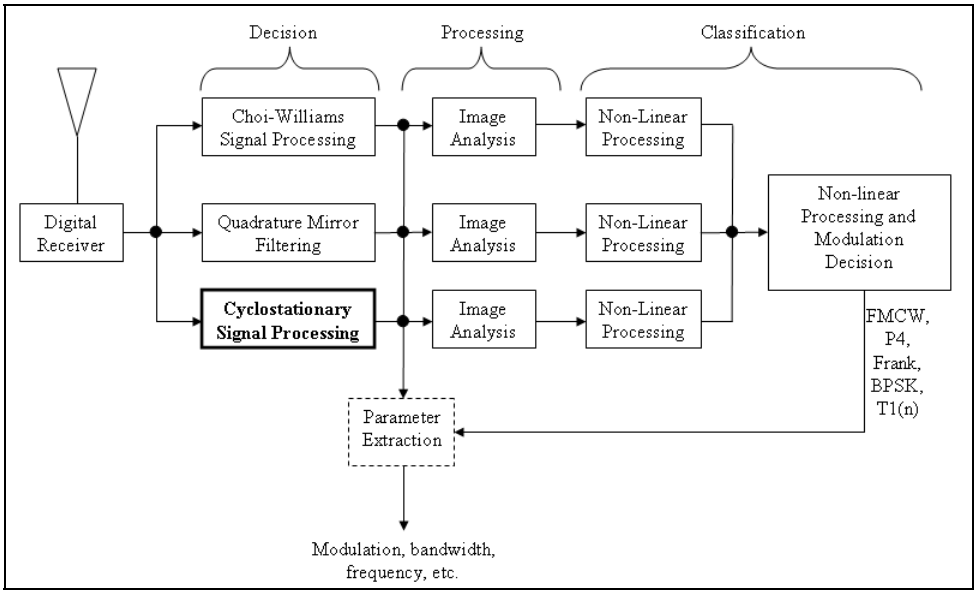

Figure 1. Electronic Intelligence Detection System (After: [3]).

B. CYCLOSTATIONARY SPECTRAL ANALYSIS

Cyclostationary spectral analysis transforms a signal into a frequency-cycle-frequency domain instead of the time-frequency domain. This represents the signal as a cyclic process rather than a stationary one which is accurate since most of the waveforms of interest are cyclic in nature. A process $x(t)$ is cyclostationary if its autocorrelation function is a periodic function of time [10]. In general, autocorrelation functions provide a measure of how closely the signal matches a copy of itself as the copy is shifted in time. Gardner and Spooner [10] define a *cyclic autocorrelation function* (involving an integral over time) which is non-zero for a set of *cycle frequencies*. If a cycle frequency or set of frequencies can be found, the process $x(t)$ is said to be cyclostationary. The *cyclic spectrum* is the Fourier transform of the cyclic autocorrelation function, and is later defined in equation (2.1). Computing the cyclic spectrum yields an estimation of the cycle frequency or frequencies of the signal in the frequency-cycle frequency domain.

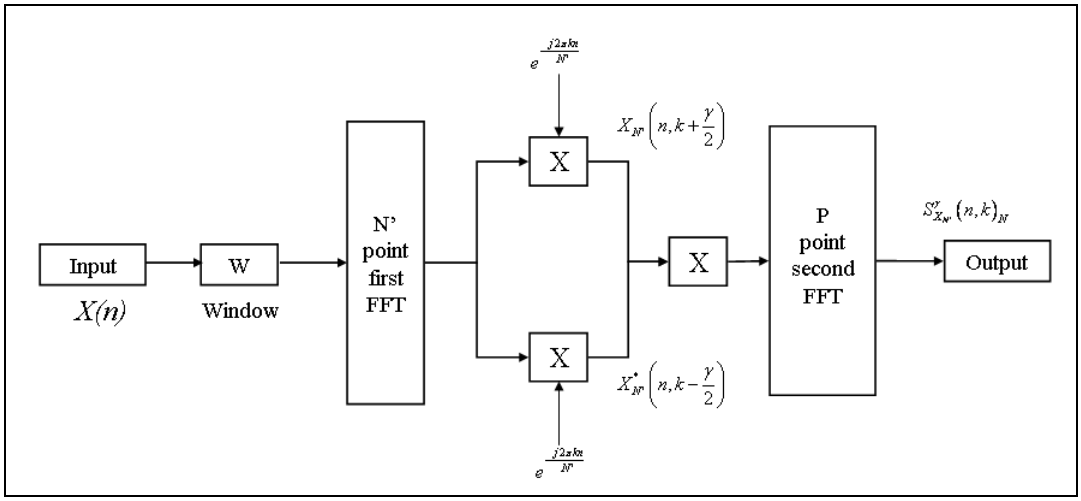

Figure 2. Block Diagram for the Cyclostationary Time-Smoothing Fast Fourier Transform Accumulation Method (From: [3]).

There are several different cyclostationary algorithms that are computationally efficient [3]. This thesis focused on the time-smoothing fast Fourier transform (FFT) accumulation method (FAM) [10]. A block diagram of the FAM is given in Figure 2. The discrete output $S^{\gamma}_{X_{N'}}(n,k)$ is estimated by:

$$S^{\gamma}_{X_{N'}}(n,k) = \frac{1}{N}\sum_{n=0}^{N-1}\left[\frac{1}{N'}X_{N'}\left(n,k+\frac{\gamma}{2}\right)X^{*}_{N'}\left(n,k-\frac{\gamma}{2}\right)\right] \tag{2.1}$$

where

$$X_{N'}(n,k) \triangleq \sum_{n=0}^{N'-1} w(n)x(n)e^{-(j2\pi kn)/N'} \tag{2.2}$$

and

k is the frequency (discrete)

N is the total number of discrete samples in the observation

N' is the number of points in the discrete (sliding) FFT

$w(n)$ is the windowing function, typically a Hamming window

$x(n)$ is the sampled complex-valued signal

$X^{*}_{N'}$ denotes the complex conjugate of $X_{N'}$

γ is the cycle frequency (discrete) (also called the frequency separation)

This algorithm divides the cycle-frequency plane into smaller regions, called channel pairs, and computes the frequency estimates one region at a time using the fast Fourier transform. The Fourier transform $X_{N'}$ of the sampled signal $x(n)$ is performed by (2.2). The cyclic spectrum (2.1) is estimated by multiplying $X_{N'}$ by its complex conjugate and integrating (summing) the result over the sample time. The final result contains N^2 small regions in the cycle-frequency plane. More details on the calculation of (2.1) and (2.2) are in Chapter III where the implementation of the algorithm is discussed.

C. SRC RECONFIGURABLE COMPUTER DESCRIPTION

1. Overview

The main hardware component used for this thesis is the SRC-6 reconfigurable computer manufactured by SRC Computers Incorporated of Colorado Springs, Colorado.

The SRC computer consists of a microprocessor, the Multi-Adaptive Processing (MAP) board, and a custom software environment called Carte developed by SRC Computers to program the system.

2. MAP Board Description

The MAP board, diagrammed in Figure 3, is SRC's explicitly controlled reconfigurable processor. Explicit control of the memory and the data processing increases the efficiency of execution and as a result increases the overall computing power of the system. Computations on the SRC computer, in theory, are more efficient when compared to standard computer architectures. The main components of interest are the memory and the field programmable gate arrays (FPGAs, labeled as 'User Logic' 1 and 2 in Figure 3). The MAP contains eight banks of dual ported memory totaling 64 megabytes (MB). Multiple banks offer the advantage of higher bandwidth memory access. The banks are connected to the board controller and the FPGAs through high speed data buses. The general purpose input output (GPIO) ports allow direct access to the data in the FPGAs.

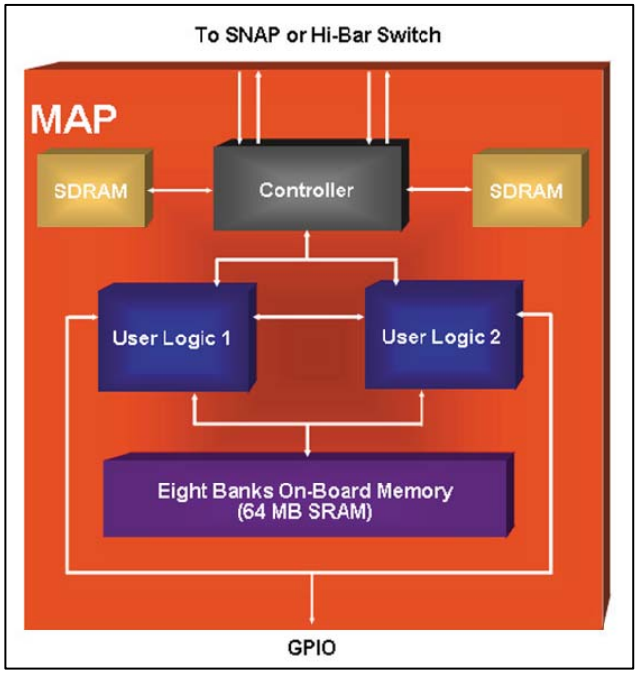

Figure 3. SRC-6 Multi-Adaptive Processing Board Architecture (From [11]).

The two Xilinx Virtex II Pro XC2VP100 Platform FPGAs are what allow the SRC-6 to be a reconfigurable computer. A generic FPGA structure is shown in Figure 4. User-specific applications can be programmed into the FPGAs, allowing the SRC-6 to be used for an unlimited number of applications. Each FPGA can communicate to the other as well as the rest of the MAP. If several MAPs are connected together, each FPGA can also communicate with the FPGAs on other MAPs. More information on the FPGA structure of the SRC-6 is found in [8]. Several parts of the cyclostationary algorithm were processed using the MAP board. More details are given in Chapter IV.

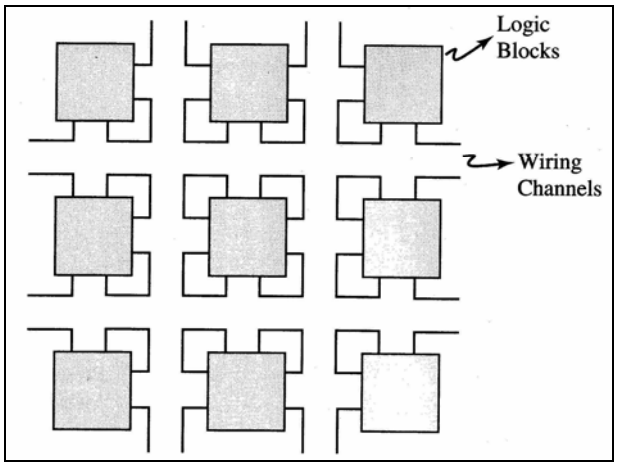

Figure 4. Generic Field Programmable Gate Array (FPGA) Structure (From [12]).

3. Carte Software Environment

Carte is the software environment for the SRC-6. The environment allows programmers to utilize the SRC-specific applications and performance advantages for programs written in standard high level programming languages. Most programs written in standard C will run on the SRC-6 but the full potential of the system will not be realized. Dedicated functions, called macros, are included in Carte and are optimized for use on the SRC-6. When the macros are used to streamline the original code, execution performance will be enhanced on the SRC-6 over standard C code. The programming guide for the SRC-6 [13] details the available macros and discusses how to use them to their fullest potential on the SRC-6.

THIS PAGE INTENTIONALLY LEFT BLANK

III. PRELIMINARY CYCLOSTATIONARY IMPLMENTATIONS

The cyclostationary FAM algorithm was initially created as a MATLAB routine included in [3] as a part of the LPI Toolbox. The LPI Toolbox is a set of MATLAB routines and scripts developed in [3] to study LPI waveforms and methods of detecting them. Inputs to the FAM algorithm are digital samples from the in-phase and quadrature channels (I- and Q-channels) of the receiving system, the sampling frequency f_s, the frequency resolution df, and a parameter M which describes the Grenander's Uncertainty Condition [14]. These inputs were maintained throughout the development of the final SRC-6 specific algorithm. In order to maintain a consistent comparison between the algorithms and the different types of signals used, a 128 Hertz (Hz) frequency resolution was used with M set to two for this thesis.

The first step was to convert the MATLAB routine to the standard C programming language. Since MATLAB is a matrix-based software tool, many of the matrix algebra routines inherent to MATLAB were recreated in C. In many cases, loops and nested loops were used to replace the functions in the original MATLAB code. The algorithm steps through the block diagram shown in Figure 2 and performs the computations outlined in (2.1) and (2.2).

A. INITIALIZATION AND INPUT CHANNELIZATION

The first part of the algorithm limits the number of input data points and creates the channel pairs mentioned in Chapter II, B. The number of input data points, N, is limited due to the computational complexity involved in large data samples. When the FAM algorithm is employed in an LPI detection system as designed, there are at least two different options for running it as data comes into the detection system:

- wait for N samples, process them, and then wait for another N samples, or
- wait for N samples, process them, and process upon receiving each additional sample (sliding FAM implementation)

N is a function of the sampling frequency, frequency resolution, and M. The data from the I-channel are divided into a matrix. The columns of this matrix become the channel pairs discussed earlier. The data from the Q-channel are discarded in the current implementation. Table 1 below shows a limited output data set from each algorithm. Note that both the Matlab and C code results are identical to six decimal places.

Table 1. MATLAB and C Channelization Comparison Data

MATLAB Column 1	**C-Generated Data Column 1**	**MATLAB Column 2**	**C-Generated Data Column 2**
0.574958	0.574958	-0.166750	-0.166750
0.601122	0.601122	-0.779386	-0.779386
-1.701171	-1.701171	-0.776057	-0.776057
-0.967194	-0.967194	0.474311	0.474311
-1.640585	-1.640585	2.326520	2.326520

The data sample in this chapter and the chapters to follow are from the analysis of a Frank code modulated signal with a 1000 Hz carrier frequency, 7000 Hz sample frequency, 64 phase codes and two cycles-per-phase. The signal data were generated using the LPI Toolbox with a 0 dB signal to noise ratio. Refer to [3] for more information on this signal.

B. WINDOWING AND FIRST FAST FOURIER TRANSFORM

Finite sampling of continuous functions gives rise to spectral leakage (non-zero values at incorrect frequencies) in the Fourier transform of the sampled function. Windowing the Fourier transform can reduce the spectral leakage and make the results more accurate. A Hamming window is applied to the data in the cyclostationary algorithm to reduce the effects of cycle and spectral leakage. In MATLAB, the hamming() function was used, but in the C algorithm the following equation was used. See [15] for more information on Hamming windows.

$$H[i] = 0.54 - 0.46\cos\left(\frac{2\pi i}{n-1}\right), \quad i = 0,1,...,n-1 \tag{2.3}$$

where

H is a column vector of an n-point symmetric Hamming window

Each row of the channelization matrix is multiplied by the corresponding row of the Hamming window.

The windowed matrix is sent through an N_P-point FFT where N_P is the number of rows in the matrix. An FFT algorithm suggested by [16] was implemented in C by Dr. Squire [17] and used with permission. Results from the FFT were carefully compared to the results from MATLAB to ensure the algorithm was working and implemented correctly. Table 2 shows a data set at this point from each algorithm (same signal as before). Note the results are still comparable.

Table 2. MATLAB and C First Transform Comparison Data

MATLAB Real Part	**C-Generated Data Real Part**	**MATLAB Imaginary Part**	**C-Generated Data Imaginary Part**
2.155325	2.155325	0.000000	0.000000
-0.728710	-0.728710	-4.195271	-4.195271
-1.403608	-1.403608	1.660082	1.660082
-0.760957	-0.760957	-0.672474	-0.672474
0.305077	0.305077	-0.012574	-0.012574

As shown in Table 2, the result from the FFT is a matrix of complex numbers. A structure called “complex” was developed in C in order to more easily track the results in the algorithm. Additionally, several functions for manipulating complex variables were created to replace the one-line MATLAB commands.

C. DOWNCONVERSION AND MATRIX MANIPULATION

The FFT result is shifted in frequency (downconverted) in order to obtain the complex factors (A.K.A demodulates) $X_{N'}$ and $X^*_{N'}$ (refer back to Figure 2). These two matrices are multiplied (correlated) before going into the second FFT. The multiplication in MATLAB utilized an element-by-element multiplication by using simple multiplication commands. In C, an additional nested loop was introduced in order to perform the element-by-element arithmetic. Table 3 shows the results at the end of this step. At this stage, both the MATLAB and the C code show the same results out to the fourth decimal place. The discrepancies in the fifth and sixth decimal places will not be significant in the final results.

Table 3. MATLAB and C Downconversion Comparison Data

MATLAB Real Part	**C-Generated Data Real Part**	**MATLAB Imaginary Part**	**C-Generated Data Imaginary Part**
-1.570607	-1.570607	9.042171	9.042172
4.641483	4.641483	-7.778432	-7.778432
8.936908	8.936909	-0.165735	-0.165735
-15.045670	-15.045671	28.069055	28.069055
-29.793129	-29.793131	16.498310	16.498309

D. SECOND FAST FOURIER TRANSFORM AND OUTPUT

The result of the previous section is sent through a second FFT and shifted. Since the FFT result is complex, the magnitude is taken. The last step in the algorithm is to determine the output range on the cycle-frequency plane and only return that range of the result. In effect, this step centralizes the cycle-frequency plane about zero. The result of the algorithm is written to an ASCII text file which can be easily read by MATLAB or

any other algorithm requiring the data. Table 4 compares the output data from the two implementations. Note that the final results are identical out to at least the sixth decimal place.

Table 4. MATLAB and C Output Comparison Data

MATLAB Column 1	**C-Generated Data Column 1**	**MATLAB Column 2**	**C-Generated Data Column 2**
0.019042	0.019042	0.017412	0.017412
0.030407	0.030407	0.001353	0.001353
0.008685	0.008685	0.013420	0.013420
0.008685	0.008685	0.001353	0.001353
0.013568	0.013568	0.006892	0.006892

E. SUMMARY AND GRAPHICAL COMPARISON OF RESULTS

Tables 1 through 4 show that the two algorithms give similar results out to at least the fourth decimal place for the test signal. The discrepancies in the fifth and sixth decimal places will be shown to be insignificant. The most efficient use of these results is to graph them. Figure 5 is a plot of the final output from the original MATLAB implementation (using the same Frank code modulated signal as before). The cycle-frequency is on the horizontal (x) axis, and the carrier frequency is on the vertical (y) axis. The same plot for the output of the implementation in C is shown in Figure 6. A close inspection of these two figures will reveal that they are indeed the same. Both figures show that the cyclostationary algorithm estimated the carrier frequency (found at the center of the shape) at about 980 Hertz, which is close to the true 1000 Hz. The difference is due to the frequency resolution (128 Hz) used by the algorithms. If a finer (smaller) resolution was used, the 980 Hz estimation would be closer to the true 1000 Hz.

Confidence in the new algorithm was increased by analyzing several LPI signals with the MATLAB and C implementations, and comparing output. Appendix A contains the comparison plots and information on the test signals. When compared, the plots were

visually identical regardless of the LPI signal used. All test signals were generated using the LPI Toolbox with a 0 dB signal-to-noise ratio.

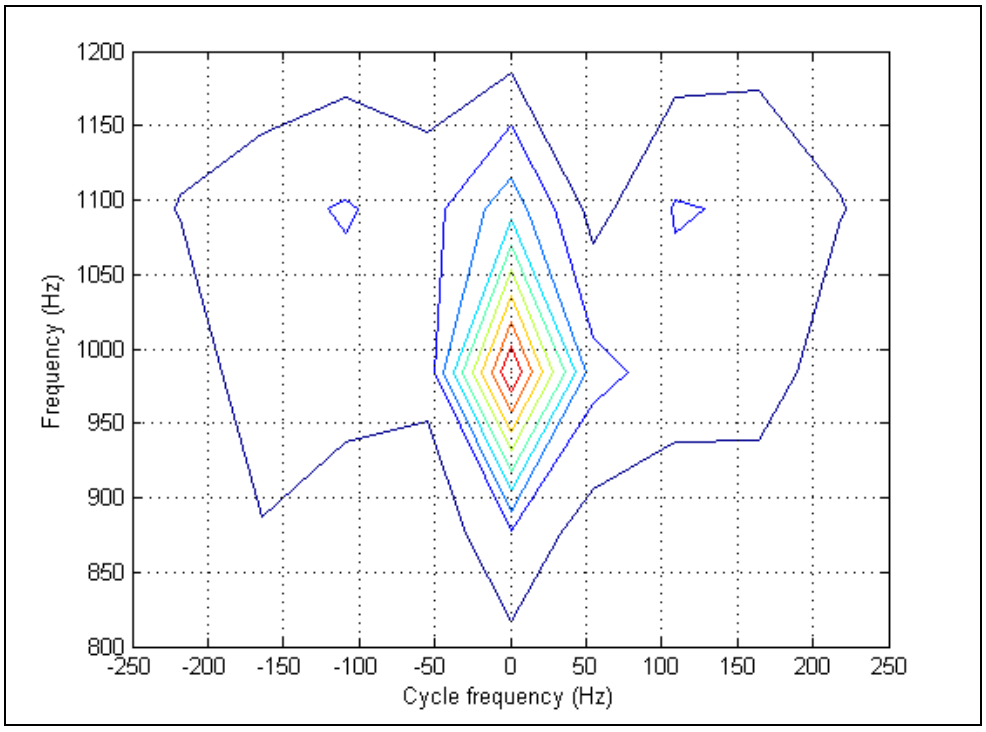

Figure 5. MATLAB Results for a Frank Code Modulated Signal.

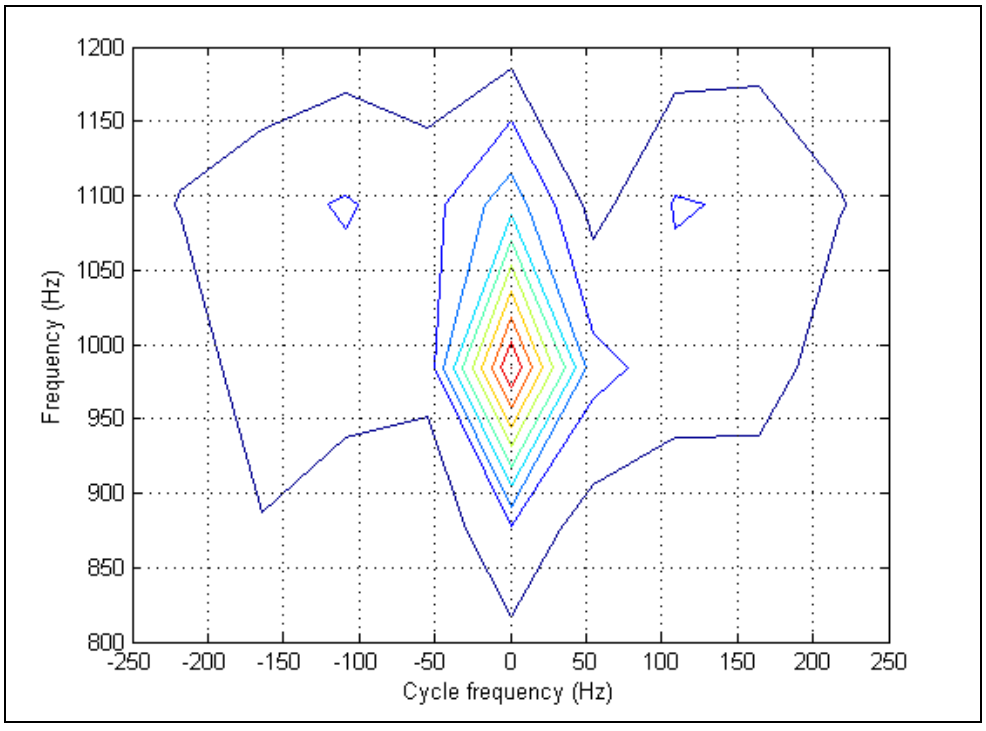

Figure 6. C Results for a Frank Code Modulated Signal.

IV. CYCLOSTATIONARY ALGORITHM ON THE SRC-6

Using the result from the previous chapter, the algorithm was moved to the SRC-6 computer and adjustments were made in order to implement the algorithm correctly. Three types of files were required to implement the algorithm: the main C file, the .mc files, and the Makefile. Any code executed on the MAP board is placed in a .mc file. The final algorithm on the SRC included three .mc files. The MakeFile is used by the SRC compiler and includes compiler and routing flags as well as file path information. Although the C programming language was used in this implementation, using Verilog or the very high speed integrated circuit hardware description language (VHDL) can have performance advantages over C.

A. INITIALIZATION AND INPUT CHANNELIZATION

The channelization code was moved over to the MAP for processing. Since there were loops which had to be created in the conversion from MATLAB to C, they were moved to the MAP to enhance performance. No adjustments in the algorithm were necessary past the SRC-specific commands to move the data back and forth between the common memory of the SRC-6 to the on-board memory (OBM) of the MAP. Table 5 shows output data from the MATLAB algorithm and the SRC algorithm. The same Frank code modulated signal from Chapter III was processed: 1000 Hz carrier frequency, 7000 Hz sample frequency, 64 phase codes and two cycles-per-phase. At this point the output is identical to at least the sixth decimal place.

Table 5. MATLAB and SRC Channelization Comparison Data

MATLAB Column 1	SRC-Generated Data Column 1	MATLAB Column 2	SRC-Generated Data Column 2
0.574958	0.574958	-0.166750	-0.166750
0.601122	0.601122	-0.779386	-0.779386
-1.701171	-1.701171	-0.776057	-0.776057
-0.967194	-0.967194	0.474311	0.474311
-1.640585	-1.640585	2.326520	2.326520

B. WINDOWING AND FIRST FAST FOURIER TRANSFORM

The same Hamming window from Chapter III was used to window the data. The two stages to generate and apply the Hamming window were combined to optimize performance on the MAP. An FFT algorithm was obtained from SRC Computers but the algorithm was not efficient for this specific application. As an alternative, the same FFT algorithm utilized in the last chapter was also considered. Both solutions are discussed below.

1. First Fast Fourier Transform Option

A drawback of the FFT algorithm from SRC Computers is that it currently requires at least 256 (2^8) points. The Frank code modulated signal that has been used so far only requires a 64-point FFT at this stage. Converting the 64 points to a 256-point array to use the FFT code is an inefficient way of utilizing resources. The pseudocode outline in Figure 7 shows how the 256-point FFT was used by padding the original array with zeros.

```
for i equals 1 to 2*256
   if i modulo (2*256/64) = 0
      put real part into x
      put imaginary part into x
   else
      put zero into x
   end if
end for

send array x to the FFT

for i equals 1 to 2*64
   put i-th FFT real and imaginary result into y
end for
```

Figure 7. Fast Fourier Transform Conversion Pseudocode.

In Figure 7, consider "x" to be the array sent to and modified by the FFT and "y" to be the array containing the final result of the FFT stage. The FFT requires that the complex data be paired [real part of signal, imaginary part of signal] in the array so the length of x going to the FFT is actually 512, with 256 real values paired with 256 imaginary values. Since there are only 64 values of real and imaginary data, the values are spaced 512/64 or eight locations apart in order to spread the 64 values over the entire array x. All other values in x are set to zero. After going through the FFT, the first 64 real and imaginary values are stored in y as the output of this stage.

Table 6 shows output from this FFT and compares it to the MATLAB algorithm. There are two discrepancies in the sixth decimal place but they were insignificant. Important to note about the FFT obtained from SRC Computers is that it is a floating point algorithm. In the LPI detection system (Figure 1), fixed point samples are received from the analog-to-digital converter. Using a fixed point FFT algorithm instead of a floating point algorithm could save computation time and memory.

Table 6. MATLAB and SRC First Transform Comparison Data: FFT 1

MATLAB Real Part	SRC-Generated Data Real Part	MATLAB Imaginary Part	SRC-Generated Data Imaginary Part
2.155325	2.155325	0.000000	0.000000
-0.728710	-0.728710	-4.195271	-4.195271
-1.403608	-1.403608	1.660082	1.660082
-0.760957	-0.760957	-0.672474	-0.672474
0.305077	0.305078	-0.012574	-0.012575

2. Second Fast Fourier Transform Option

A second FFT algorithm was considered in order to more efficiently utilize the MAP resources for smaller-point transforms. Since it did not have the 256 point minimum, it was not necessary to send an array that was at least 256 points long. Table 7 shows output from this FFT and compares it to the MATLAB algorithm. Although there are two discrepancies in the sixth decimal place of the imaginary portion of the data, it will be shown later that the same conclusions will be drawn from the final result (refer to Figure 11). Chapter IV, which compares the timing results of the algorithms, shows that the custom implementation of the FFT was not as fast as the FFT provided by SRC Computers.

Table 7. MATLAB and SRC First Transform Comparison Data: FFT 2

MATLAB Real Part	**SRC-Generated Data Real Part**	**MATLAB Imaginary Part**	**SRC-Generated Data Imaginary Part**
2.155325	2.155325	0.000000	0.000000
-0.728710	-0.728708	-4.195271	-4.195270
-1.403608	-1.403609	1.660082	1.660081
-0.760957	-0.760957	-0.672474	-0.672473
0.305077	0.305077	-0.012574	-0.012575

C. DOWNCONVERSION AND MATRIX MANIPULATION

This step was also moved over to the MAP for processing. Several adjustments were necessary past the SRC-specific ones to move the data back and forth between the common memory of the SRC-6 to the on-board memory of the MAP. The downconversion step included a multiplication by a complex exponential (cosine and sine terms). It was found that calculating the terms using the microprocessor and sending the values to the MAP as an input improved the overall performance of the algorithm. Additionally, loops were combined to more efficiently take the matrix transpose of the downconversion result. Table 8 below shows sample output data and compares it to MATLAB. Discrepancies exist in the fifth and sixth decimal places. This is simply a difference in the FFT algorithms between MATLAB and the SRC-6.

Table 8. MATLAB and SRC Downconversion Comparison Data

MATLAB Real Part	SRC-Generated Data Real Part	MATLAB Imaginary Part	SRC-Generated Data Imaginary Part
-1.570607	-1.570607	9.042171	9.042170
4.641483	4.641483	-7.778432	-7.778430
8.936908	8.936907	-0.165735	-0.165733
-15.045670	-15.045670	28.069055	28.069058
-29.793129	-29.793132	16.498310	16.498306

D. SECOND FAST FOURIER TRANSFORM AND OUTPUT

The last part of the algorithm was partially implemented on the MAP. The two FFT algorithms mentioned earlier were used again to perform the second and last transform. Experimentation showed that the code that takes the magnitude of the FFT output and centers the data about zero on the cycle-frequency plane was more efficiently implemented on the microprocessor than on the MAP. Multiple divisions are required in order to center the data, and as [11] documents, divisions on the MAP cost performance. Since many of the divisions were not dependant upon the loop indexes, an attempt was made to move the divisions outside of the loop. Experimentation showed that further performance was gained by putting the loop back on the microprocessor. Table 9 compares the output of the data from the algorithm. Note that the output is real (non-complex) and although the numbers are somewhat different, they are comparable in magnitude. Section F shows that the same conclusions are drawn when the output from the two implementations is plotted.

Table 9. MATLAB and SRC Output Comparison Data

MATLAB Column 1	**SRC-Generated Data Column 1**	**MATLAB Column 2**	**SRC-Generated Data Column 2**
0.019042	0.006565	0.017412	0.016968
0.030407	0.012845	0.001353	0.001333
0.008685	0.003793	0.013420	0.013395
0.008685	0.011983	0.001353	0.018792
0.013568	0.001181	0.006892	0.003529

E. LIMITATIONS OF THE ALGORITHM

There are several limitations of the algorithm on the SRC-6 which should be noted. First, as mentioned in Chapter III, a frequency resolution of 128 Hz was used with M set to two for this thesis. Increasing the frequency resolution or increasing M increased the size of the matrices used on the MAP board. When this was attempted, it was found that the new matrix sizes were too large to fit in the on-board memory (OBM) banks of the MAP board. Therefore, a resolution of 128 Hz with M equal to two is recommended for the sampling frequencies used in this thesis.

The second limitation concerns the array and matrix declarations within the MAP routines. The size of the arrays and matrices must be known prior to execution. Since the MAP routines all use the same constants, a single file containing the constants was generated and included in each MAP routine. The drawback is that the constants depend on the sampling frequency, frequency resolution, and the parameter M. Anytime any of these parameters are changed, the constant file must be regenerated and the hardware (the FPGAs on the MAP) must be reconfigured. It can take several hours to configure the hardware for the FAM implementation.

Closely related to both limitations above is the sampling frequency of the analog-to-digital converter (ADC). A 400 MHz sampling frequency was achieved using a Maxim ADC in [7]. At this frequency, the matrices required by the FAM algorithm are too large for the OBM memory banks of the MAP. A tradeoff exists between the sampling frequency and the frequency resolution of the algorithm: in order to raise the sampling frequency the frequency resolution must be sacrificed. A table of frequency resolutions, values of the parameter M, and the maximum sampling frequencies available is in Appendix B. The frequency resolution must be reduced to at least 3.125 MHz in order to achieve a sample frequency of 400 MHz. Maintaining the convention that the frequency resolution must be a factor of 2^X, the resolution must be reduced to 4.194 MHz (2^{22}). At this resolution, the results obtained are too ambiguous to deduce meaningful conclusions about the signal being processed.

F. SUMMARY AND GRAPHICAL COMPARISON OF RESULTS

In summary, the standard C code was transferred to the SRC-6 Computer. Some adjustments were made to the C code in order to optimize execution on the SRC-6. A custom FFT algorithm was implemented to improve the utilization of resources. Tables 6 through 9 showed that the algorithm on the SRC produces similar results to at least the sixth decimal place. Figures 8 through 10 show plots of the output from the algorithms. Figure 8, the result from MATLAB, is the same as Figure 5 and is repeated for convenience. Figure 9 shows the result from the SRC algorithm using the FFT routine provided by SRC Computers. Although the shape is slightly different in this case, the same conclusion can be drawn from both figures: the transmitted signal was approximately 1000 Hz. Figure 10 shows the result from the SRC algorithm using the custom FFT routine. The shape closely matches the MATLAB result in Figure 8 and the same conclusion as above can be drawn. A close examination of several types of LPI signals was necessary in order to ensure the results would be accurate.

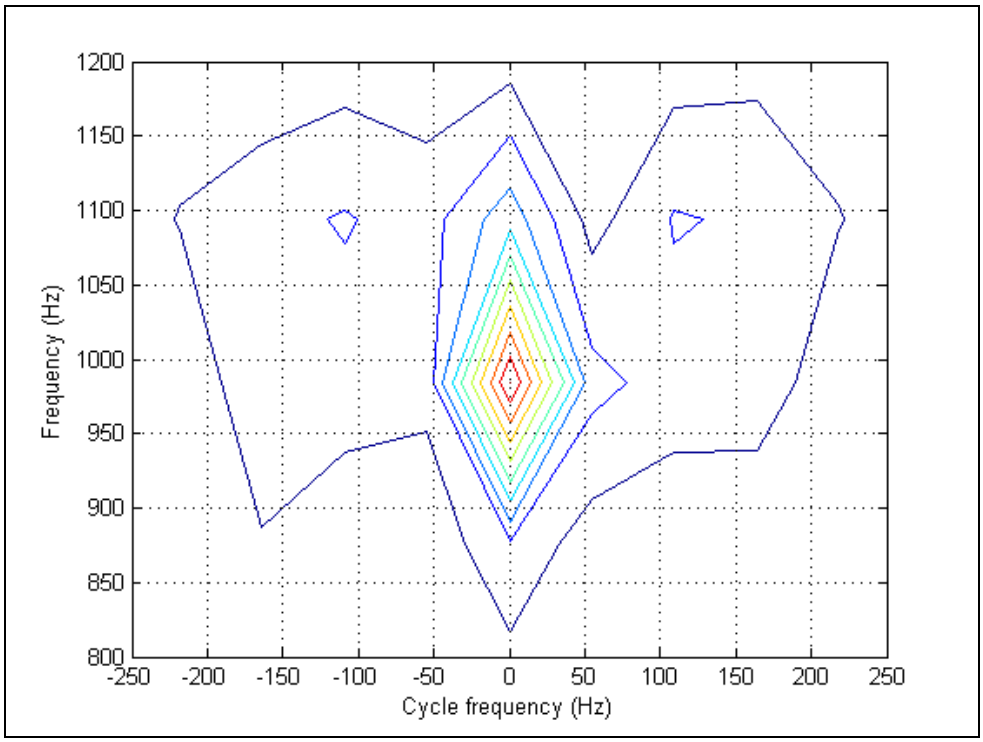

Figure 8. MATLAB Results for a Frank Code Modulated Signal.

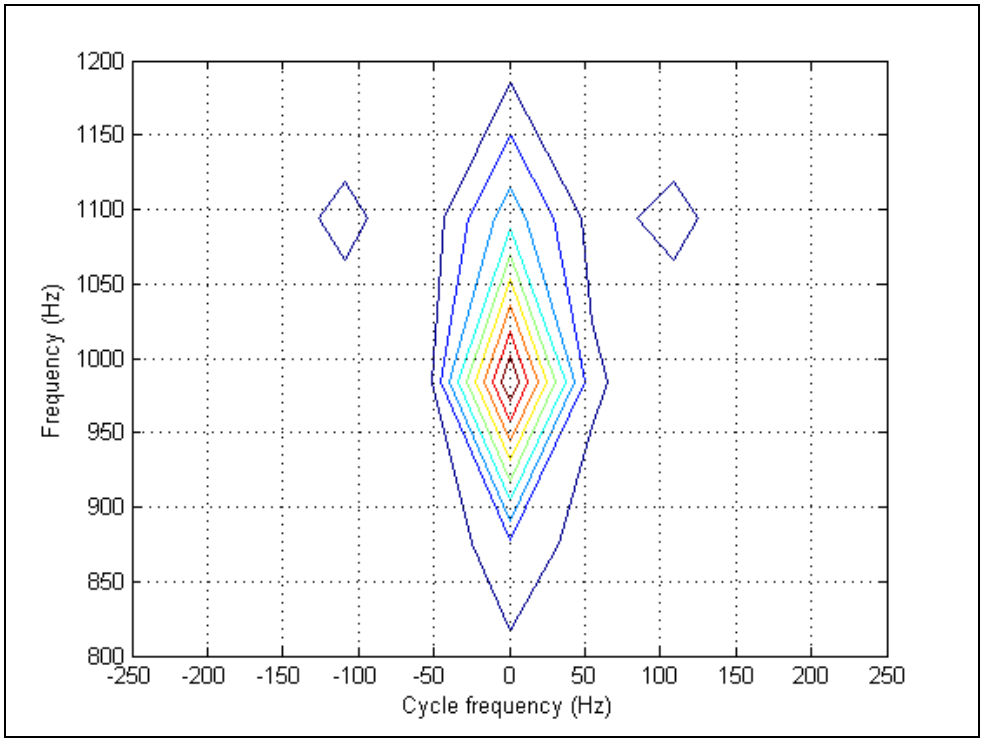

Figure 9. SRC Results for a Frank Code Modulated Signal (FFT 1).

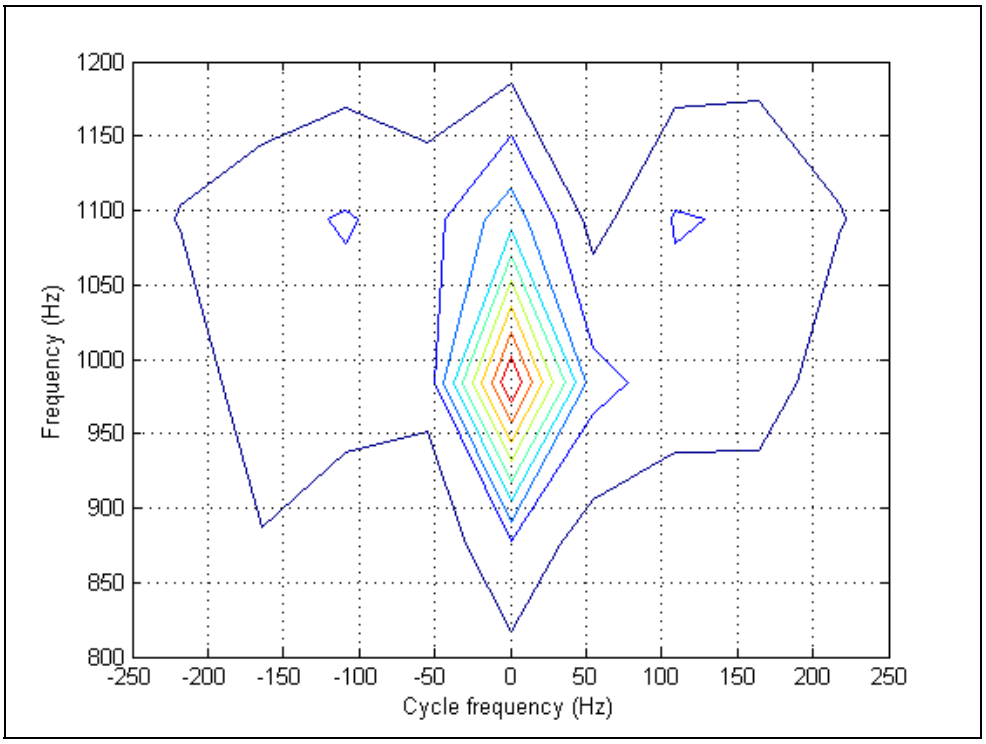

Figure 10. SRC Results for a Frank Code Modulated Signal (FFT 2).

V. TIMING AND PERFORMANCE ANALYSIS

The last stage was to compare the timing performance of all three algorithms. A timing estimate in MATLAB was provided by the `tic` and `toc` functions. The timing library in the C language was used in conjunction with the library available on the MAP board to do the timing for the C and SRC-specific algorithms. High precision timing functions were used to time execution times less than one second. The data transfers between the microprocessor and the MAP were not included in the timing results. The final algorithm on the SRC involves three different transfers between the common memory and MAP board. If a detection system like the one proposed is implemented on the SRC-6, the GPIO ports on the MAP would be used and as a result, the data would be readily available for use on the MAP without using data transfers. Additionally, in an LPI detection system the final result from the cyclostationary processing block will be sent to other blocks for processing. Since plots such as the ones presented in this thesis will not be generated, the code used to generate the plots was not included when the algorithm was timed. Note that all plots were generated in MATLAB using the same code.

Each signal was sent through the algorithms 20 times, and an average time was taken. Table 10 shows a summary of the results and Appendix C provides the detailed results of each run. The timing in MATLAB was dependant upon several factors including the particular computer system used and the number of processes running in the background of the Windows-based operating system. A 3.0 GHz Pentium 4 computer running Windows XP (service pack 2) with 2.0 GB of random access memory was used for this thesis. The C code was compiled and executed using a Linux operating system with a standard Linux C compiler (icc). Note that compiler flags were not utilized to generate the executable.

The results in Table 10 show that the timing performance of the standard C code is close to the execution time of the MATLAB algorithm. Both implementations outperform the algorithm on the SRC-6. Despite this, realization of the bigger-picture

ELINT detection system with the SRC-6 is not unreasonable since the SRC-6 timing of this algorithm is comparable to the other implementations. An experienced SRC programmer should be able to further improve the timing performance of the FAM algorithm. The execution time is on the order of a second, so without improvements, this algorithm may not be suitable for some kinds of radar warning receiver applications where the execution time needs to be more expedient.

Table 10. Average Timing Performance Results: Entire Algorithm

Test Signal	**MATLAB (seconds)**	**C[5] (seconds)**	**SRC-6[5] (General FFT) (seconds)**	**SRC-6[5] (Custom FFT) (seconds)**
Frank[1]	0.056551	0.056926	0.233520	0.251517
FMCW[2]	0.052374	0.056803	0.242449	0.250500
Costas[3]	0.170558	0.226554	0.621600	0.650096
FSK/PSK Costas[4]	0.173712	0.225923	0.615036	0.645937

[1]Frank code modulated signal with 1000 Hz carrier frequency, 7000 Hz sampling frequency, 64 phase codes and two cycles per phase.

[2]Frequency modulated continuous wave signal with 1000 Hz carrier frequency, 7000 Hz sampling frequency, 250 Hz modulation frequency and a 20 millisecond modulation period.

[3]Costas coded signal where the codes are [3000, 2000, 6000, 4000, 5000, 1000] Hz with a 5 millisecond duration and 15,057 Hz sampling frequency.

[4]Frequency- and Phase- shift keying combination technique with the same Costas sequence as above.

[5]The C and SRC-6 timing purposely did not include the time to transfer the data from the microprocessor board to the MAP board.

In addition to the overall results, the execution time of only the FFT portion of the algorithm was compared. Table 11 shows that the C and the SRC algorithms (with the FFT provided by SRC) were approximately equal. Matlab was the quickest, and the SRC implementation with the custom FFT algorithm was the slowest. Execution of the channelization and downconverstion stages were also timed but the results were insignificant given the time spent performing the FFTs.

Table 11. Average Timing Performance Results: FFT Only

Test Signal	**MATLAB (seconds)**	**C (seconds)**	**SRC-6 (General FFT) (seconds)**	**SRC-6 (Custom FFT) (seconds)**
Frank	0.002811	0.022224	0.022654	0.075691
FMCW	0.002247	0.022233	0.022654	0.075691
Costas	0.009225	0.088476	0.090484	0.300416
FSK/PSK Costas	0.008978	0.088561	0.090484	0.300416

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSIONS AND RECOMMENDATIONS

A. SUMMARY AND CONCLUSIONS

This thesis expanded the ongoing investigation of the feasibility of implementing an ELINT detection system on an SRC-6 reconfigurable computer. Reconfigurable computers have been suggested as a solution to real-time processing of LPI ELINT radar signals. In this thesis, a cyclostationary spectral analysis algorithm was successfully implemented on the SRC-6 and showed comparable performance to equivalent algorithms in MATLAB and standard C. Several LPI signals were tested to demonstrate the robustness of the algorithm and compare processing times required by each of the signals. Based on the signals tested, the cyclostationary analysis can be performed in approximately one second by the SRC-6. The cyclostationary algorithm can be used to aid in the detection, identification, and classification of LPI signals. The ELINT detection system as a whole could be used in a theater environment to provide near-real-time battlefield information and situational awareness.

B. RECOMMENDATIONS FOR FUTURE WORK

The performance of the cyclostationary spectral analysis algorithm developed on the SRC-6 could be enhanced by optimizing execution on the MAP board and the microprocessor. In the MATLAB routine, it was not necessary to optimize performance, hence extra variables and loose coding methods were used to ease readability of the MATLAB code. In the derivation to the SRC-6 algorithm, some portions of the code were improved, but the introduction of nested loops and extra matrix manipulations offset any advantage gained. Streamlining the code could improve timing performance and result in quicker parameter extraction.

There are several types of cyclostationary spectral analysis [3]. This thesis focused on the time-smoothing FFT accumulation method, but there is also the direct frequency-smoothing method which could be implemented and incorporated into the LPI detection system.

This thesis is one block of the ELINT detection system shown in Figure 1. Although several of the blocks have already been implemented in other theses, they do not currently work as part of a single system. Future work should include a conversion from the individual blocks to a unified detection system. Optimizing individual parts, such as the cyclostationary algorithm, does not necessarily mean the system as a whole will be optimized when assembled. Therefore, future work should also include optimization of the detection system as a whole.

APPENDIX A. SIGNAL DETAILS AND PLOT COMPARISONS

A summary of the signals used in this thesis is below. All signals were generated by the LPI Toolbox and were generated with a 0 dB signal-to-noise ratio. Most of the signals below were selected to accommodate a parallel comparison between this thesis and the Choi-Williams algorithm [2]. More information about the LPI Toolbox and each signal below are provided in [3].

- Frank code modulated signal with a 1000 Hz carrier frequency, 7000 Hz sampling frequency, 64 phase codes, and two cycles per phase. This was the signal analyzed in detail in Chapters III and IV.
- Frequency modulated continuous wave modulated signal with a 1000 Hz carrier frequency, 7000 Hz sampling frequency, 250 Hz modulation bandwidth, and a 20 millisecond modulation period.
- Costas coded signal with a 15,057 Hz sampling frequency, Costas sequence of [3000, 2000, 6000, 4000, 5000, 1000] Hz with a five millisecond duration.
- Frequency shift keying (FSK) and phase shift keying (PSK) combination technique with the same Costas sequence: [3000, 2000, 6000, 4000, 5000, 1000] Hz with a five millisecond duration.

Plots of the FAM results from the MATLAB and the C algorithms are below. Since the Frank code was used in Chapters III and IV, refer to Figures 5 and 6 to compare those two plots. The plots are comparable for each algorithm. It should also be noted that in general, cyclostationary processing does not work well on the Costas and FSK/PSK Costas modulations due to the lack of time information. As a result, Figures 13 through 16 do not compare to the results shown in Figures 11 and 12.

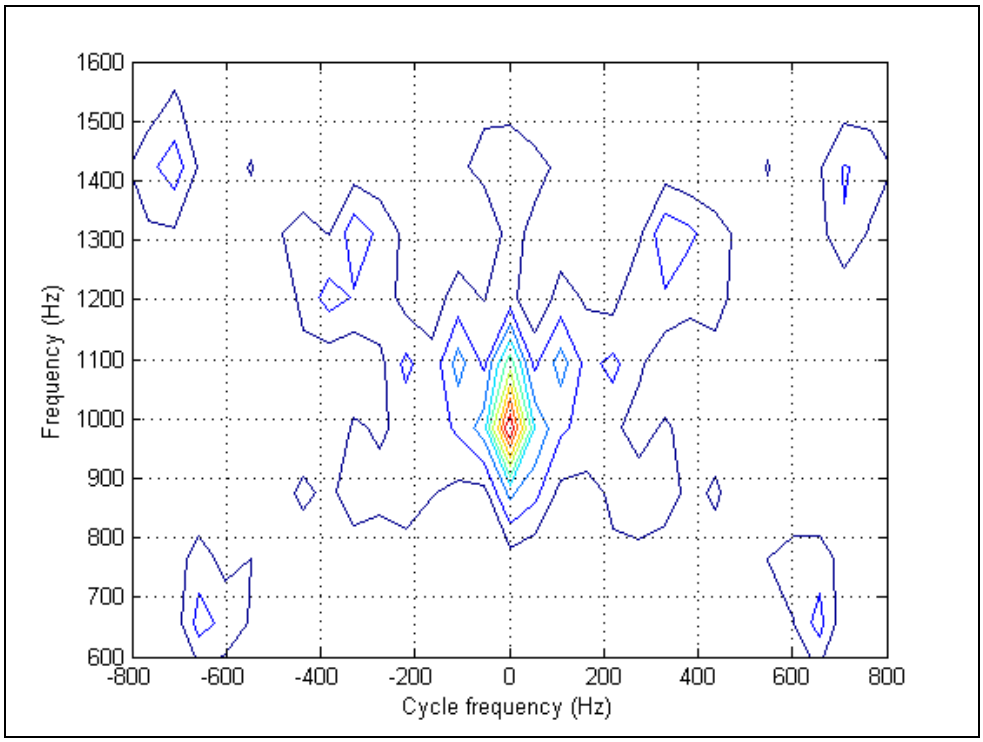

Figure 11. MATLAB Results for a Frequency Modulated Continuous Wave Signal.

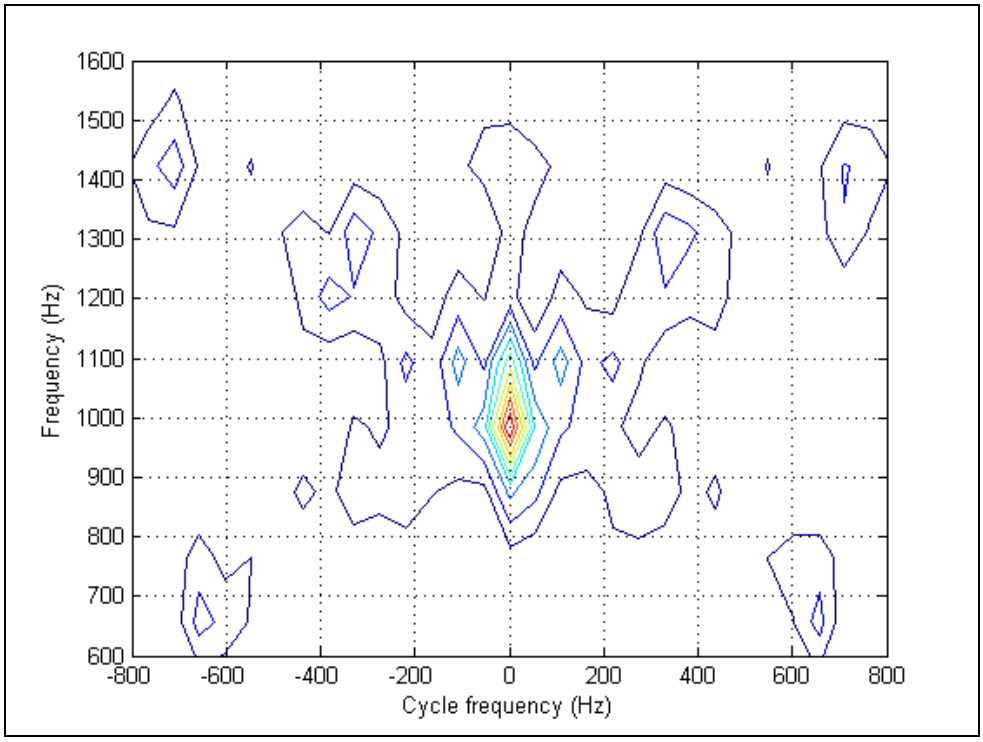

Figure 12. C Results for a Frequency Modulated Continuous Wave Signal.

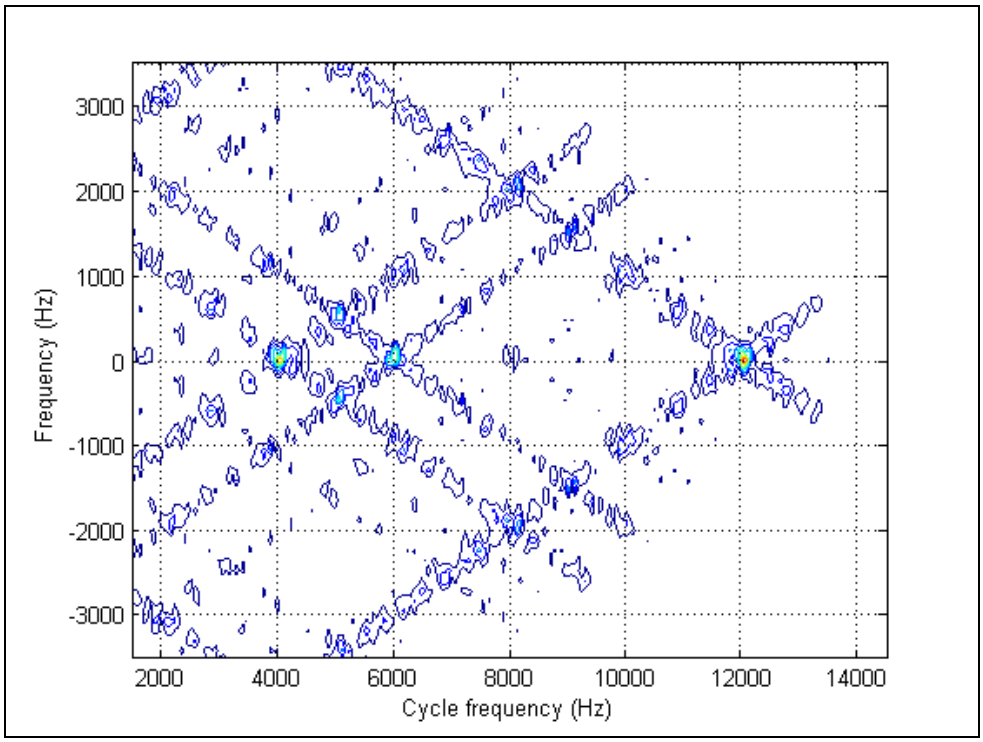

Figure 13. MATLAB Results for a Costas Coded Signal.

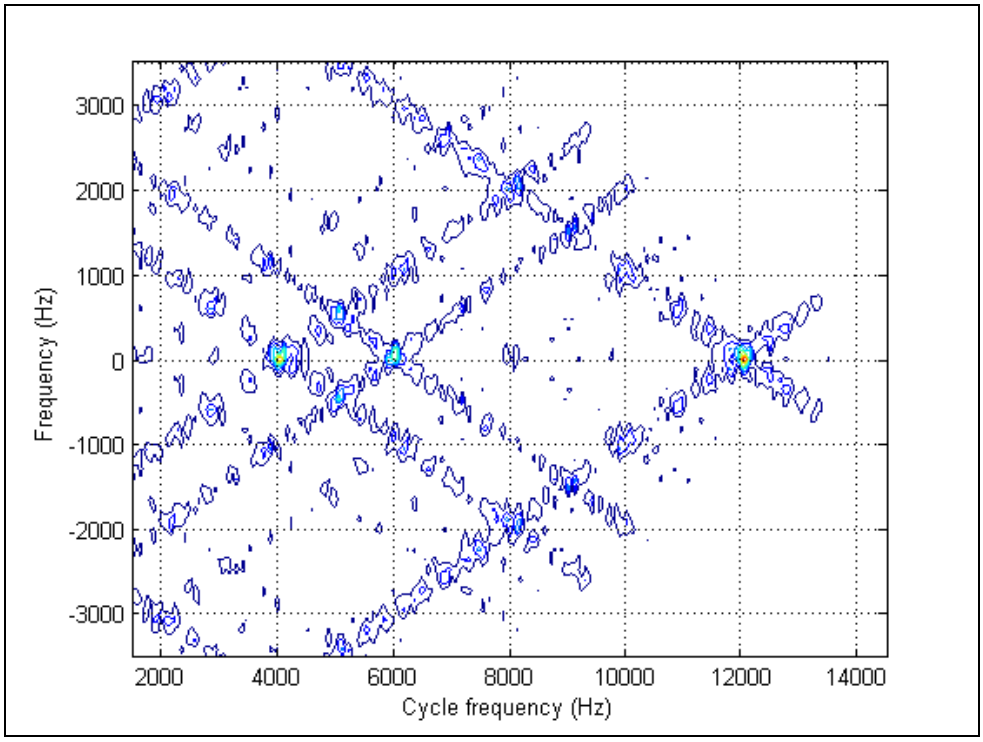

Figure 14. C Results for a Costas Coded Signal.

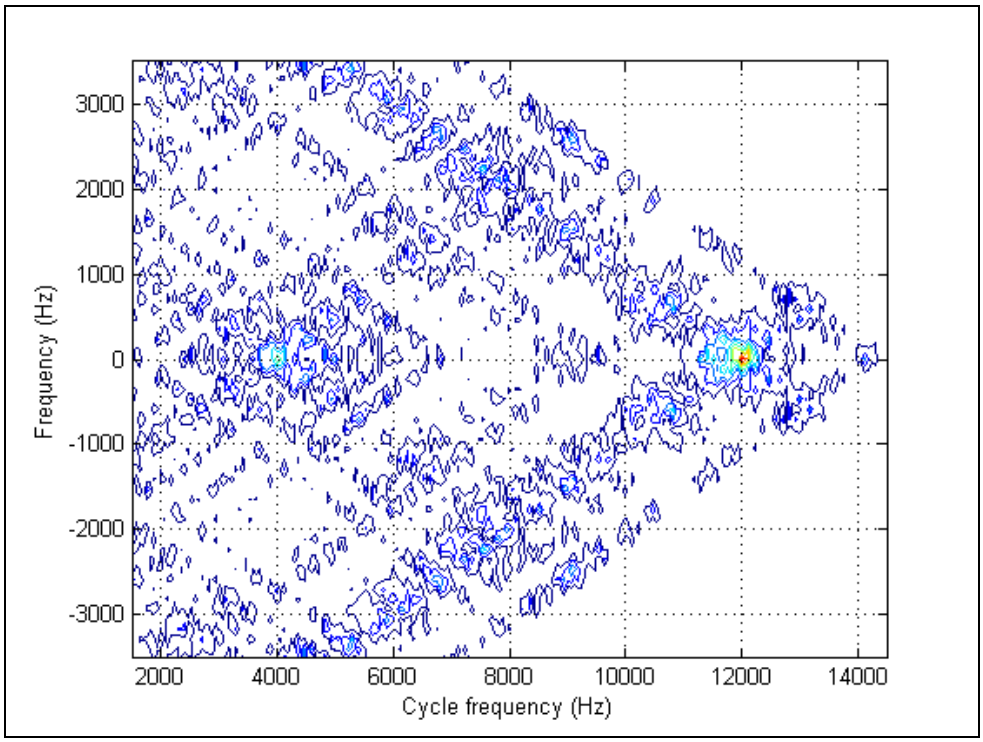

Figure 15. MATLAB Results for a FSK/PSK Costas Coded Signal.

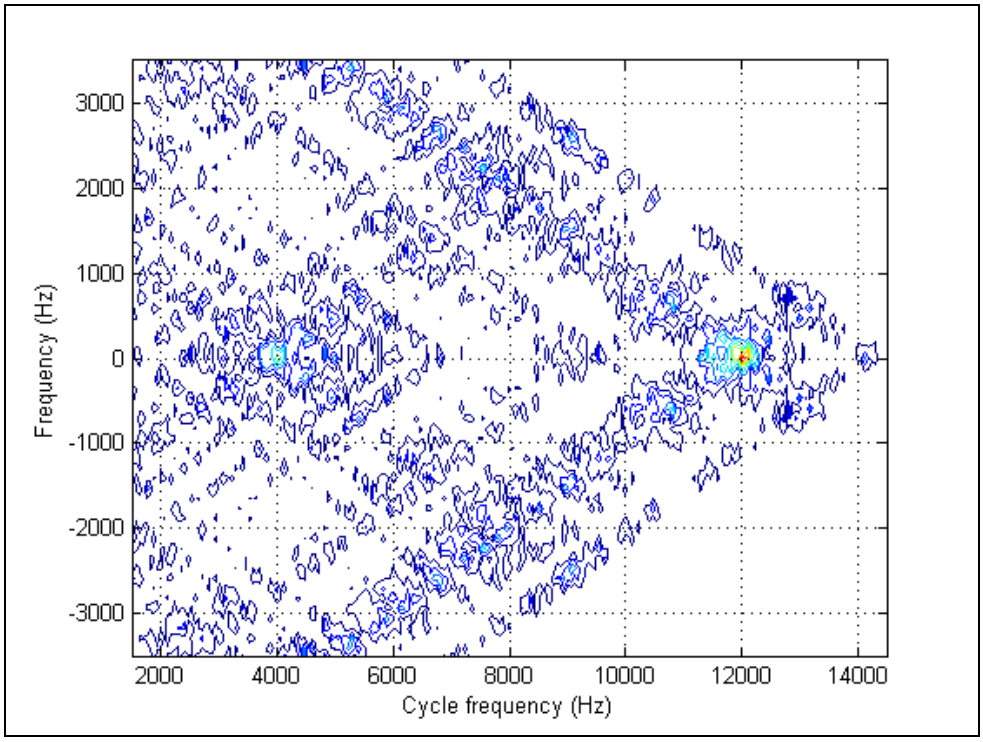

Figure 16. C Results for a FSK/PSK Costas Coded Signal.

APPENDIX B. SAMPLE FREQUENCY LIMITATION DATA

Chapter IV, E discusses how the frequency resolution used in the FAM algorithm can result in matrix sizes which exceed the available memory space of MAP board on the SRC-6. The next two sections to follow show the data which led to this conclusion and the MATLAB script used to generate the data.

A. SAMPLE FREQUENCY LIMITATION CALCULATIONS

Table 12 shows the results of the calculations. The sampling frequency noted is the maximum available before the matrices become too large to fit into a single OBM bank.

Table 12. Sample Frequency Limitation Results

Frequency Resolution (Hz)		***M***	**Sampling Frequency (Hz)**	**Matrix Size (elements)**
x	2^x			
1	2	2	257	131072
		4	257	262144
		8	129	131072
		12	129	262144
2	4	2	513	131072
		4	513	262144
		8	257	131072
		12	257	262144
3	8	2	1025	131072
		4	1025	262144
		8	513	131072
		12	513	262144
5	32	2	4097	131072
		4	4097	262144
6	64	2	8193	131072
		4	8193	262144
7	128	2	16385	131072
		4	16385	262144
8	256	2	32769	131072
		4	32769	262144
9	512	2	65537	131072
		4	65537	262144
10	1024	2	131073	131072

Table 12. Sample Frequency Limitation Results

Frequency Resolution (Hz)		***M***	**Sampling Frequency (Hz)**	**Matrix Size (elements)**
x	2^x			
		4	131073	262144
11	2048	2	262145	131072
		4	262145	262144
21	2.097e6	2	268.436e6	131072
		4	268.436e6	262144
21.5754	3.125e6	2	400.0e6	131072
		4	400.0e6	262144
22	4.194e6	2	400.099e6	131072
		4	400.099e6	262144

B. SAMPLE FREQUENCY LIMITATION MATLAB CODE

Below is the MATLAB script that was used to generate the data in Table 12. A brute-force method was used so execution takes some time. The parameter *MAX_OBM_SIZE* on the MAP board describes the maximum number of 64 bit elements that a single OBM bank can hold. The variables df, fs_start, dfs, and fs_end can be changed to target specific sampling frequencies and decrease the execution time.

```
% file name: MaxFs_test
% Matlab .m file to determine max possible sample frequency for a
% given frequency resolution.
% Max possible frequency is determined in a loop by increasing the
% sample frequency and determining how large (how many elements) will
% be in the resulting data matrices.

% Declare Maximum number of 64 bit elements in the MAP's OBM
   MAX_OBM_SIZE = 523776;

% Declare 1. the frequency resolution, df, in Hertz, and
%         2. the starting frequency, fs_start, for thisfs (lower
%            bound on sampling frequency search)
%         3. the rate at which the sampling frequency is increased,
%            dfs.
%         4. the upper bound on the sampling frequency search
% Note 1: all can be changed to target specific frequency areas.
% Note 2: all should be carefully chosen: the more calculations
%         performed, the longer it will take.  Example: if it is
%         known that a high frequency is targeted, fs_start should
%         be large and dfs should be set accordingly until the range
%         is narrowed down.
   % Use these for lower log2(n) values
```

```
      %df       = 2.^[1:10];
      %fs_start = 1;
      %dfs      = 1;  % increase by 1 Hz every time
      %fs_end   = 400.1e6; %400.1 MHz
   % Use these for the first 2^n crossover of fs = 100 MHz
      %df       = 2.^[19:20];
      %fs_start = 99.5e6; % 99.5 MHz
      %dfs      = 2;  % increase by 2 Hz every time
      %fs_end   = 400.1e6; %400.1 MHz
   % Use these for the first ever crossover of fs = 100 MHz
      %df       = 781248:1:781251;
      %fs_start = 99.5e6; % 99.5 MHz
      %dfs      = 2;  % increase by 2 Hz every time
      %fs_end   = 100.1e6; %100.1 MHz
   %Use these for the first 2^n crossover of fs = 400 MHz
      %df       = 2.^[21:22];
      %fs_start = 268.0e6; % 268 MHz
      %dfs      = 2;  % increase by 2 Hz every time
      %fs_end   = 400.1e6; %400.1 MHz
   % Use these for the first ever crossover of fs = 400 MHz
      df       = 3124999:1:3125002; %~3.125 MHz
      fs_start = 399.7e6; % 399.7 MHz
      dfs      = 2;  % increase by 2 Hz every time
      fs_end   = 400.1e6; %400.1 MHz
   % Use these to fully regenerate the data in Table 11
   % Note: Takes a while (~ hours) to execute.
      %df = [2.^[1:3, 5:11, 21], 3125000, 2^22];
      %fs_start = 0;
      %dfs      = 1;  % increase by 1 Hz every time
      %fs_end   = 40010000; %400.1 MHz

% Declare M
% Note: if higher M is not of interest, don't use it to cut down on
%       execution time.  In general, higher M leads to lower max
%       sample frequencies.
   M = [2 4 8 12];

tic;
count = 1; % output variable index
for ii = 1:max(size(df))
   for jj = 1:max(size(M))
      dalpha = df(ii)/M(jj);
      prev_num_el = 0;
      num_el = 0;
      thisfs = fs_start;
      while (num_el  <= MAX_OBM_SIZE) & (thisfs < fs_end)
         prev_num_el = num_el; % keep track of previous # of elements
         Np     = pow2(nextpow2(thisfs/df(ii)));
         L      = Np/4;
         P      = pow2(nextpow2(thisfs/dalpha/L));

         num_el = Np*Np*P; % number of elements in biggest matrix
         thisfs = thisfs + dfs; % increase sample frequency
      end
      df_out(count) = df(ii); % freq resolution out
```

```
      M_out(count)  = M(jj); % M out
      fs_out(count) = thisfs - dfs; % max sample freq for df and M
      num_el_out(count)  = prev_num_el; % resulting number of elements
      count = count + 1;

      clc
      disp([num2str(ii*100/max(size(df))), '% through df vector']);
      disp([df_out', M_out', fs_out', num_el_out'])
   end
end

clc, pause(1)
fprintf('          df           M          fs     Num Elements\n')
disp([[df_out'], M_out', [fs_out', num_el_out']])

disp('Script to find maximum sample frequency is done executing.')
if ~isempty(find(num_el_out == 0))
   disp(['Zeros in the number of elements column indicate that ', ...
         'the starting frequency was too high.'])
end
toc
disp(['   or ', num2str(toc/60), ' minutes.'])
disp(['Finish Time: ', datestr(now)])
beep
```

APPENDIX C. TIMING RESULTS

Tables 13 through 20 show the detailed timing results for 20 samples. "SRC-6 V1" refers to the algorithm implemented using the FFT routines provided by SRC Computers, Inc. "V2" refers to the custom FFT algorithm discussed in Chapter IV, B section 2.

Table 13. Frank Code Modulation Overall Timing Results

Trial	MATLAB	C	SRC-6 V1	SRC-6 V2
1	0.065983	0.056536	0.244003	0.252773
2	0.054135	0.058750	0.242305	0.252130
3	0.053511	0.056458	0.244351	0.250380
4	0.069488	0.056799	0.244978	0.251912
5	0.044624	0.057075	0.248450	0.252143
6	0.044004	0.056622	0.246449	0.253331
7	0.054468	0.056728	0.242674	0.252173
8	0.065375	0.056543	0.247886	0.251224
9	0.059834	0.056646	0.241619	0.249693
10	0.045817	0.057625	0.024328	0.251018
11	0.078991	0.056869	0.244638	0.250818
12	0.045630	0.057064	0.244995	0.253746
13	0.081227	0.056878	0.243615	0.250794
14	0.057429	0.057022	0.246451	0.252454
15	0.051130	0.056832	0.245741	0.252777
16	0.053611	0.056652	0.242520	0.250342
17	0.054519	0.056693	0.241712	0.249933
18	0.045804	0.057191	0.241529	0.249731
19	0.052619	0.056459	0.246717	0.251450
20	0.052829	0.057071	0.245448	0.251516
Mean	0.056551	0.056926	0.233520	0.251517
Minimum	0.044004	0.056458	0.024328	0.249693
Maximum	0.081227	0.058750	0.248450	0.253746
Std. Dev.	0.010758	0.000516	0.049282	0.001186

Table 14. Frank Code Modulation FFT Timing Results

Trial	MATLAB	C	SRC-6 V1	SRC-6 V2
1	0.002271	0.022263	0.022654	0.075691
2	0.002238	0.022107	0.022654	0.075691
3	0.002172	0.022191	0.022654	0.075691
4	0.002645	0.022367	0.022654	0.075691
5	0.002589	0.022264	0.022654	0.075691
6	0.002020	0.022098	0.022654	0.075691
7	0.011239	0.022205	0.022654	0.075691
8	0.002580	0.022239	0.022654	0.075691
9	0.002373	0.022199	0.022654	0.075691
10	0.002530	0.022236	0.022654	0.075691
11	0.002104	0.022255	0.022654	0.075691
12	0.002616	0.022227	0.022654	0.075691
13	0.002042	0.022137	0.022654	0.075691
14	0.002132	0.022225	0.022654	0.075691
15	0.002883	0.022153	0.022654	0.075691
16	0.002519	0.022229	0.022654	0.075691
17	0.002896	0.022328	0.022654	0.075691
18	0.002107	0.022318	0.022654	0.075691
19	0.002127	0.022239	0.022654	0.075691
20	0.002128	0.022201	0.022654	0.075691
Mean	0.002811	0.022224	0.022654	0.075691
Minimum	0.002020	0.022098	0.022654	0.075691
Maximum	0.011239	0.022367	0.022654	0.075691
Std. Dev.	0.002003	0.000069	0.000000	0.000000

Table 15. FMCW Signal Overall Timing Results

Trial	MATLAB	C	SRC-6 V1	SRC-6 V2
1	0.067238	0.057316	0.232175	0.243699
2	0.044698	0.056422	0.245989	0.250032
3	0.050839	0.056425	0.247112	0.249807
4	0.072432	0.056960	0.243245	0.249713
5	0.053013	0.056810	0.241738	0.249848
6	0.045374	0.056805	0.242486	0.251349
7	0.054124	0.056923	0.233073	0.240814
8	0.043416	0.056805	0.234867	0.252723
9	0.043906	0.056530	0.245740	0.250727
10	0.053206	0.057304	0.239342	0.253731
11	0.052723	0.056571	0.245592	0.251417
12	0.058227	0.056601	0.243599	0.248947
13	0.044371	0.056769	0.241409	0.251275
14	0.054510	0.057108	0.244187	0.251457
15	0.053961	0.056942	0.242189	0.251851
16	0.043406	0.056579	0.245674	0.252781
17	0.050171	0.056419	0.245446	0.250861
18	0.043970	0.057177	0.246327	0.253736
19	0.059011	0.056664	0.244162	0.254293
20	0.058880	0.056925	0.244637	0.250932
Mean	0.052374	0.056803	0.242449	0.250500
Minimum	0.043406	0.056419	0.232175	0.240814
Maximum	0.072432	0.057316	0.247112	0.254293
Std. Dev.	0.008070	0.000281	0.004379	0.003203

Table 16. FMCW Signal FFT Timing Results

Trial	MATLAB	C	SRC-6 V1	SRC-6 V2
1	0.002153	0.022381	0.022654	0.075691
2	0.002359	0.022230	0.022654	0.075691
3	0.002095	0.022168	0.022654	0.075691
4	0.002309	0.022443	0.022654	0.075691
5	0.002136	0.022124	0.022654	0.075691
6	0.003054	0.022304	0.022654	0.075691
7	0.002121	0.022237	0.022654	0.075691
8	0.002094	0.022203	0.022654	0.075691
9	0.002132	0.022173	0.022654	0.075691
10	0.002144	0.022168	0.022654	0.075691
11	0.002134	0.022263	0.022654	0.075691
12	0.002186	0.022276	0.022654	0.075691
13	0.002139	0.022116	0.022654	0.075691
14	0.002120	0.022182	0.022654	0.075691
15	0.002685	0.022231	0.022654	0.075691
16	0.002346	0.022239	0.022654	0.075691
17	0.002155	0.022232	0.022654	0.075691
18	0.002321	0.022268	0.022654	0.075691
19	0.002152	0.022193	0.022654	0.075691
20	0.002097	0.022233	0.022654	0.075691
Mean	0.002247	0.022233	0.022654	0.075691
Minimum	0.002094	0.022116	0.022654	0.075691
Maximum	0.003054	0.022443	0.022654	0.075691
Std. Dev.	0.000237	0.000079	0.000000	0.000000

Table 17. Costas Code Modulated Signal Overall Timing Results

Trial	MATLAB	C	SRC-6 V1	SRC-6 V2
1	0.161000	0.225959	0.713059	0.718018
2	0.186860	0.226425	0.617260	0.647683
3	0.187260	0.224767	0.620602	0.641515
4	0.166050	0.226218	0.614482	0.645447
5	0.167770	0.225187	0.617364	0.646697
6	0.171440	0.224945	0.616011	0.644498
7	0.159100	0.226241	0.619140	0.647419
8	0.179070	0.226327	0.616288	0.645562
9	0.182980	0.228498	0.615274	0.645542
10	0.180250	0.226358	0.621412	0.648578
11	0.164730	0.234440	0.615774	0.646587
12	0.166990	0.225468	0.615571	0.647125
13	0.149600	0.227953	0.621919	0.648038
14	0.160990	0.225347	0.616050	0.644420
15	0.174670	0.224905	0.617053	0.648282
16	0.167320	0.226212	0.618521	0.648496
17	0.168170	0.224695	0.605365	0.645980
18	0.190430	0.226369	0.615798	0.650374
19	0.166610	0.226559	0.618737	0.643961
20	0.159860	0.228207	0.616325	0.647704
Mean	0.170558	0.226554	0.621600	0.650096
Minimum	0.149600	0.224695	0.605365	0.641515
Maximum	0.190430	0.234440	0.713059	0.718018
Std. Dev.	0.010888	0.002151	0.021793	0.016111

Table 18. Costas Code Modulated Signal FFT Timing Results

Trial	MATLAB	C	SRC-6 V1	SRC-6 V2
1	0.008391	0.088262	0.090484	0.300416
2	0.008301	0.088738	0.090484	0.300416
3	0.008560	0.088680	0.090484	0.300416
4	0.009028	0.088678	0.090484	0.300416
5	0.008795	0.088328	0.090484	0.300416
6	0.009164	0.088406	0.090484	0.300416
7	0.008684	0.088454	0.090484	0.300416
8	0.009441	0.088366	0.090484	0.300416
9	0.008585	0.088334	0.090484	0.300416
10	0.008895	0.088396	0.090484	0.300416
11	0.008423	0.088541	0.090484	0.300416
12	0.008987	0.088651	0.090484	0.300416
13	0.008639	0.088648	0.090484	0.300416
14	0.009227	0.088758	0.090484	0.300416
15	0.008746	0.088595	0.090484	0.300416
16	0.008962	0.088230	0.090484	0.300416
17	0.008465	0.088390	0.090484	0.300416
18	0.008902	0.088331	0.090484	0.300416
19	0.008535	0.088299	0.090484	0.300416
20	0.017769	0.088437	0.090484	0.300416
Mean	0.009225	0.088476	0.090484	0.300416
Minimum	0.008301	0.088230	0.090484	0.300416
Maximum	0.017769	0.088758	0.090484	0.300416
Std. Dev.	0.002034	0.000169	0.000000	0.000000

Table 19. FSK/PSK Costas Code Modulated Signal Overall Timing Results

Trial	MATLAB	C	SRC-6 V1	SRC-6 V2
1	0.202210	0.225113	0.605364	0.635203
2	0.154820	0.225982	0.614785	0.649715
3	0.158900	0.226223	0.613092	0.645732
4	0.214220	0.226064	0.618733	0.647635
5	0.166440	0.226394	0.616445	0.647799
6	0.168840	0.225918	0.615726	0.638406
7	0.170220	0.226804	0.615682	0.651529
8	0.161380	0.225892	0.605148	0.644833
9	0.165620	0.226364	0.615092	0.649002
10	0.159330	0.225368	0.618790	0.644401
11	0.168010	0.224938	0.615257	0.642788
12	0.240720	0.226686	0.616989	0.649306
13	0.166940	0.225076	0.618617	0.645718
14	0.165360	0.224818	0.614285	0.646938
15	0.174800	0.226221	0.615275	0.647153
16	0.168080	0.225191	0.617026	0.644288
17	0.159760	0.225044	0.614572	0.648736
18	0.167120	0.227099	0.614903	0.646868
19	0.181050	0.225876	0.617404	0.645819
20	0.160410	0.227387	0.617536	0.646865
Mean	0.173712	0.225923	0.615036	0.645937
Minimum	0.154820	0.224818	0.605148	0.635203
Maximum	0.240720	0.227387	0.618790	0.651529
Std. Dev.	0.021377	0.000750	0.003693	0.003790

Table 20. FSK/PSK Costas Code Modulated Signal FFT Timing Results

Trial	MATLAB	C	SRC-6 V1	SRC-6 V2
1	0.008296	0.088476	0.090484	0.300416
2	0.008351	0.088764	0.090484	0.300416
3	0.009029	0.088542	0.090484	0.300416
4	0.009065	0.088527	0.090484	0.300416
5	0.008971	0.088582	0.090484	0.300416
6	0.009297	0.088723	0.090484	0.300416
7	0.008575	0.088536	0.090484	0.300416
8	0.009138	0.088518	0.090484	0.300416
9	0.009121	0.088501	0.090484	0.300416
10	0.009130	0.089153	0.090484	0.300416
11	0.008761	0.088846	0.090484	0.300416
12	0.009135	0.088392	0.090484	0.300416
13	0.009042	0.088172	0.090484	0.300416
14	0.009332	0.088659	0.090484	0.300416
15	0.009138	0.088342	0.090484	0.300416
16	0.009130	0.088756	0.090484	0.300416
17	0.008740	0.088696	0.090484	0.300416
18	0.009090	0.088288	0.090484	0.300416
19	0.009077	0.088458	0.090484	0.300416
20	0.009142	0.088286	0.090484	0.300416
Mean	0.008978	0.088561	0.090484	0.300416
Minimum	0.008296	0.088172	0.090484	0.300416
Maximum	0.009332	0.089153	0.090484	0.300416
Std. Dev.	0.000286	0.000226	0.000000	0.000000

APPENDIX D. C CODE FOR THE ALGORITHM

A. C MAIN PROGRAM

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include "complex_subrs.c"
#include "other_subrs.c"
#include "fft.c"
#include "high_prec_time.c"

#define pi 3.141592653589793

FILE *I_ptr; // pointer to the I-channel input file name

FILE *IFFT1_Out;  // pointer to the I-channel output file
                  // name for the first FFT results
FILE *QFFT1_Out;  // pointer to the Q-channel output file
                  // name for the first FFT results
FILE *IFFT2_Out;  // pointer to the I-channel output file
                  // name for the second FFT results
FILE *QFFT2_Out;  // pointer to the Q-channel output file
                  // name for the second FFT results

FILE *Output;     // pointer to the final output file name

struct complex complex_exp(double A);
struct complex complex_mult(struct complex A,
                            struct complex B);
struct complex complex_conj(struct complex A);
float complex_mag(struct complex A);

float rem(float A, float B);

int main()
{


   /* DECLARE VARIABLES AND CONSTANTS */
      /* declare file names and path */
         char I_file[] = "I_channel.txt";

         char IFFT1_Out_file[] = "IFFT1_out.txt";
         char QFFT1_Out_file[] = "QFFT1_out.txt";
         char IFFT2_Out_file[] = "IFFT2_out.txt";
         char QFFT2_Out_file[] = "QFFT2_out.txt";

         char Output_file[] = "FAM_result.txt";
```

```
      /* Declare Input Variables */
         int fs = 15057; // sample frequency
         int df = 128;  // frequency resolution
         int M = 2;   // M = df/alpha

      /* Declare all timing variables and get first time
         hac; start timing */
         struct timeval t0, t1, t2, t3, time1, time2;
         struct timeval dummy_time2, subr_t0, subr_t1;
         float total_fft_time = 0.0, fft_only_time = 0.0, dummy_time1 =
0.0;
         float cum_execution_time = 0.0;
         gettimeofday(&t0, NULL);
         int timei;

      /* Calculate dalpha */
         double dalpha = df/M;
      /* determine number of input channels: ds/df */
         double Np = pow(2.0, ceil(log10(fs/df) /
            log10(2)) );
      /* Overlap factor in order to reduce the number of
         short time fft's. L is the offset between points
         in the same column at consecutive rows. L shoud
         be less than or equal to Np/4
         (Prof. Loomis paper) */
         double L = Np/4;
      /* determine number of columns formed in the
         channelization matrix (x) */
         double P = pow(2.0, ceil(log10(fs/dalpha/L) /
            log10(2)) );
      /* determine total number of points in the input
         data to be processed */
         double N = P*L;

      /* declare other variables and arrays to be used
         Note: I tried to declare them in the order in
         which they are needed. Some were consolidated.*/
         /* loop indexes */
            int i=0, j=0, k=0, index=0;
         /* Array to contain values from input file */
            float I_Values[(int)N];
         /* Initial Array and Matrix */
            double NN = (P-1)*L+Np; //resizes x into X
            double x[(int)NN];
            double X[(int)Np][(int)P];
            int xInitialMax; //book-keeping on x array
         /* Need to store Hamming Window */
            double hamming[(int)Np];
            double XW[(int)Np][(int)P];  // windowed X
         /* Variables Specific to the First FFT */
            double *fft1_in,   *fft1_ot; // in and out
            double *fft1_scr1, *fft1_scr2; // scratch
            fft1_in   = malloc(2 * Np * sizeof(double));
            fft1_ot   = malloc(2 * Np * sizeof(double));
```

```
            fft1_scr1 = malloc(2 * Np * sizeof(double));
            fft1_scr2 = malloc(2 * Np * sizeof(double));
            struct complex temp_XF1[(int)Np][(int)P];
         /* FFT 1 Shift and Downconversion Variables */
            struct complex XF1[(int)Np][(int)P];
            struct complex E[(int)Np][(int)P];
            struct complex XD[(int)Np][(int)P];
            struct complex XE[(int)P][(int)Np];
            struct complex XM[(int)P][(int)Np*(int)Np];
         /* Variable Specific to the Second FFT */
            double *fft2_in,   *fft2_ot; // in and out
            double *fft2_scr1, *fft2_scr2; // scratch
            fft2_in   = malloc(2 * P * sizeof(double));
            fft2_ot   = malloc(2 * P * sizeof(double));
            fft2_scr1 = malloc(2 * P * sizeof(double));
            fft2_scr2 = malloc(2 * P * sizeof(double));
            struct complex temp_XF2[(int)P][(int)(Np*Np)];
         /* FFT 2 shift and Matrix Manipulation */
            struct complex XF2[(int)P][(int)(Np*Np)];
         /* Magnitude of FFT 2 Results */
            float MM[(int)((3*P/4)-(P/4))+1][(int)(Np*Np)];
         /* Final Output Matrix */
            float Sx[(int)Np+1][2*(int)N+1];
         /* Data Display Variables */
            float c, p, alpha, f, kk, ll, Sx_max;

       /* get second time hac: stop timing.  Pulling data
          in does not count against timing. */
         gettimeofday(&t1, NULL);
         timei = timeval_subtract(&dummy_time2, &t1, &t0);
         cum_execution_time        +=        dummy_time2.tv_sec        +
dummy_time2.tv_usec*1e-6;

   /* OPEN THE INPUT FILE */
      I_ptr = fopen(I_file, "r");
      if (I_ptr==NULL)
      {
         printf("Error opening I-channel input file.\n");
         return(1);
      }

   /* READ IN THE I-CHANNEL FILE */
      /* use the next two lines to test the first
         value of the file
         fscanf(I_ptr, "%f", &I_Values[0]);
         printf("The first value in the input file
         is %1.16f\n", I_Values[0]);
      */
      /* This while loop reads in the entire file and puts
         it into the I_Values array */
         while ( (fscanf(I_ptr, "%f",
                  &I_Values[i]) != EOF) && (i<N) )
         {
            x[i] = I_Values[i];
            i++;
```

```
         }
      /* This if and while loop fills the X array with
         zeros if there wasn't N rows of data */
         if (i < N)
            while (i < N)
            {
               x[i] = 0;
               i++;
            }
         xInitialMax = i;

         fclose(I_ptr);

       /* get time hac; restart timing */
         gettimeofday(&t0, NULL);

/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   INPUT CHANNELIZATION - this part limits the total
          number of points to be analyzed. It also
          generates a Np-by-P matrix, X, with shifted
          versions of the input vector in each column.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
   /* Zero fill x if we don't have NN Samples. The loop
      does the xx(NN) = 0 loop in the Matlab code. */
      for(i=xInitialMax; i<NN;i++)  x[i] = 0;

   for (i=0; i<P; i++)
   {
      index = 0;
      for (j=i*(int)L+1; j<=i*(int)L+Np; j++)
      {
        X[index][i] = x[j-1];
        index++;
      }
   }

/* The following loop was used to generate data for G.
   Upperman's Thesis */
printf("******** THESIS DATA: CHANNELIZATION *****\n");
    for (i=0; i<5; i++)
   {
      printf("\t");
      for (j=0; j<2; j++)
      {
            printf("%2.8f\t", X[i][j]);
      }
      printf("\n");
   }

/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   HAMMING WINDOW - a vector of length Np is created with
          the hamming function (below main) then this
          vector is inserted in a Np X Np matrix diagonal
          and this result is multiplied by the chennlized
          input matrix (x).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
   for (i=0; i<Np; i++)
   {
      hamming[i] = 0.54 - 0.46*cos(2*pi*i/(Np-1));
   }

   /* The loops below apply the Hamming Window
      (they do the XW=diag... command in the Matlab
      version) */
      for (i=0; i<Np; i++)
      {
         for (j=0; j<P; j++)
         {
            XW[i][j] = hamming[i]*X[i][j];
         }
      }

/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   FIRST FFT CALL
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
   for(i=0; i<P; i++) //col
   {
      j = 0;
      for(k=0; k<Np; k++) //row
      {
         fft1_in[j + 0] = XW[k][i];
         fft1_in[j + 1] = 0;
         j+=2;
      } // for k

      /* Get time hac and call FFT */
         gettimeofday(&t1, NULL);
         timei = timeval_subtract(&dummy_time2, &t1, &t0);
         cum_execution_time        +=        dummy_time2.tv_sec        +
dummy_time2.tv_usec*1e-6;

         gettimeofday(&subr_t0, NULL);
            fft(fft1_in,    fft1_ot,    fft1_scr1,    fft1_scr2,    Np,
&dummy_time1);
         gettimeofday(&subr_t1, NULL);

         fft_only_time += dummy_time1;
         timei = timeval_subtract(&dummy_time2, &subr_t1, &subr_t0);
         total_fft_time         +=         dummy_time2.tv_sec         +
dummy_time2.tv_usec*1.0e-6;
         cum_execution_time += dummy_time1;
         gettimeofday(&t0, NULL);

      j = 0;
      for(k=0; k<Np; k++)
      {
         temp_XF1[k][i].x = fft1_ot[j + 0];
         temp_XF1[k][i].y = fft1_ot[j + 1];
         j+=2;
      } // for k
```

```
   } // for i

   /* PRINT FFT1 OUTPUT FILE
      IFFT1_Out=fopen(IFFT1_Out_file, "w");
      if(IFFT1_Out == NULL)
      {
         puts("Error creating FFT 1 output file.");
         return(1);
      }

      QFFT1_Out=fopen(QFFT1_Out_file, "w");

      for(i=0; i<Np; i++)
      {
         for(j=0; j<P; j++)
         {
            fprintf(IFFT1_Out, "%6.32e\t",
               temp_XF1[i][j].x);
            fprintf(QFFT1_Out, "%6.32e\t",
               temp_XF1[i][j].y);
         }
         fprintf(IFFT1_Out, "\n");
         fprintf(QFFT1_Out, "\n");
      }
      fclose(IFFT1_Out);
      fclose(QFFT1_Out);
   */

/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   FFT SHIFT - implements the FFT shift and
           left/right flip in the matlab code in one
           single loop.  End result is that the top and
           bottom halves of the fft are swapped.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
   for(i=0; i<(Np/2); i++)
   {
      for(j=0; j<P; j++)
      {
         // Real bottom half becomes real top half:
            XF1[i][j].x = temp_XF1[i+(int)(Np/2)][j].x;
         // Real top half becomes bottom real half:
            XF1[i+(int)(Np/2)][j].x = temp_XF1[i][j].x;

         // Imag bottom half becomes imag top half:
            XF1[i][j].y = temp_XF1[i+(int)(Np/2)][j].y;
         // Imag top half becomes imag bottom half:
            XF1[i+(int)(Np/2)][j].y = temp_XF1[i][j].y;
      }
   }

/* The following loop was used to generate data for G.
   Upperman's Thesis */
   printf("******* THESIS DATA: FFT 1 AND SHIFT ****\n");
   for (i=0; i<5; i++)
   {
```

```
      printf("\t%2.8f+i*%2.8f\n", XF1[i][0].x,
         XF1[i][0].y);
   }

/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   Downconversion - the short sliding FFT's results are
   shifted to baseband to obtain decimated complex
   demodulate sequences
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
   for(i=0; i<Np; i++)
   {
      k = i-((int)Np/2);
      for(j=0; j<P; j++)
      {
         E[i][j] = complex_exp(-2*pi*k*j*L/Np);
         XD[i][j] = complex_mult(XF1[i][j], E[i][j]);
      }
   }

/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   MATRIX TRANSPOSE
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
      for (i=0; i<Np; i++)
      {
         for (j=0; j<P; j++)
         {
            XE[j][i].x = XD[i][j].x;
            XE[j][i].y = XD[i][j].y;
         }
      }

/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   MULTIPLICATION - the product sequences between each
          one of the complex demodulates and the complex
          conjugate of the others are formed.  This forms
          the area in the bi-frequency plane.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
      for (i=0; i<Np; i++)
      {
         for (j=0; j<Np; j++)
         {
            for (k=0; k<P; k++)
            {
               XM[k][i*(int)Np+j] =
                  complex_mult(XE[k][i],
                  complex_conj(XE[k][j]));
            } // for k
         } // for j
      } // for i

/* The following loop was used to generate data for G.
   Upperman's Thesis */
   printf("****** THESIS DATA: Downconversion ****\n");
   for (i=0; i<5; i++)
   {
```

```
         printf("\t%2.8f+i*%2.8f\n", XM[i][1].x,
            XM[i][1].y);
   }

/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   SECOND FFT - a P point FFT is applied to XM (in each
          of its columns)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
   for(i=0; i<Np*Np; i++) //col
   {
      j = 0;
      for(k=0; k<P; k++) //row
      {
         fft2_in[j + 0] = XM[k][i].x;
         fft2_in[j + 1] = XM[k][i].y;
         j+=2;
      }

      /* get time hac and call fft */
         gettimeofday(&t1, NULL);
         timei = timeval_subtract(&dummy_time2, &t1, &t0);
         cum_execution_time        +=        dummy_time2.tv_sec        +
dummy_time2.tv_usec*1e-6;

         gettimeofday(&subr_t0, NULL);
            fft(fft2_in,    fft2_ot,    fft2_scr1,    fft2_scr2,    P,
&dummy_time1);
         gettimeofday(&subr_t1, NULL);

         fft_only_time += dummy_time1;
         timei = timeval_subtract(&dummy_time2, &subr_t1, &subr_t0);
         total_fft_time         +=         dummy_time2.tv_sec         +
dummy_time2.tv_usec*1.0e-6;
         cum_execution_time += dummy_time1;
         gettimeofday(&t0, NULL);
      j = 0;
      for(k=0; k<P; k++)
      {
         temp_XF2[k][i].x = fft2_ot[j + 0];
         temp_XF2[k][i].y = fft2_ot[j + 1];
         j+=2;
      }
   }

   /* PRINT FFT1 OUTPUT FILE
      IFFT2_Out=fopen(IFFT2_Out_file, "w");
      if(IFFT2_Out == NULL)
      {
         puts("Error creating FFT 2 output file.");
         return(1);
      }

      QFFT2_Out=fopen(QFFT2_Out_file, "w");

      for(i=0; i<P; i++)
```

```
      {
         for(j=0; j<Np*Np; j++)
         {
            fprintf(IFFT2_Out, "%6.32e\t", temp_XF2[i][j].x);
            fprintf(QFFT2_Out, "%6.32e\t", temp_XF2[i][j].y);
         }
         fprintf(IFFT2_Out, "\n");
         fprintf(QFFT2_Out, "\n");
      }
      fclose(IFFT2_Out);
      fclose(QFFT2_Out);
   */

/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   MATRIX MANIPULATION - implements the FFT shift and
          left/right flip in the matlab code in one
          single loop.  End result is that the top and
          bottom halves of the fft are swapped.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
   for(i=0; i<=((P/2)-1); i++)
   {
      for(j=0; j<Np*Np; j++)
      {
         // Real bottom half becomes real top half:
            XF2[i][j].x = temp_XF2[i+(int)(P/2)][j].x;
         // Real top half becomes bottom real half:
            XF2[i+(int)(P/2)][j].x = temp_XF2[i][j].x;

         // Imag bottom half becomes imag top half:
            XF2[i][j].y = temp_XF2[i+(int)(P/2)][j].y;
         // Imag top half becomes imag bottom half:
            XF2[i+(int)(P/2)][j].y = temp_XF2[i][j].y;
      }
   }

   /* Obtain the magnitude of the complex values */
      for(i=(P/4)-1; i<(3*P/4); i++)
      {
         for(j=0; j<Np*Np; j++)
         {
            MM[i-(int)(P/4)+1][j] =
               complex_mag(XF2[i][j]);
         }
      }

/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   DATA DISPLAY - display only the data inside the range
          of interest - centralizes the bi-frequency
          plane according to alpha0 and f0 vectors.
          Note: the alpha0 and f0 vectors are defined
          as follows (in matlab terms):
             alpha0 = -fs  :fs/N :fs;
             f0     = -fs/2:fs/Np:fs/2;
          but are not declared in this program since
          they are only used for plotting the results.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
   Sx_max = 0;
   for(i=0; i<=P/2; i++)
   {
      for(j=0; j<Np*Np; j++)
      {
          if(rem(j+1, Np) == 0)
          {
             c = .5*Np - 1;
          }
          else
          {
             c = rem(j+1, Np) - .5*Np - 1;
          }

          k = ceil((j+1)/Np) - .5*Np - 1;
          p = i - .25*P;
          alpha = ((k-c)/Np) + ((p-1)/N);
          f = (k+c)/(2*Np);

          if (((alpha > -1) & (alpha < 1)) |
             ((f >-.5)      & (f < .5)))
          {
             kk = 1+Np*(f + .5);
             ll = 1+N*(alpha + 1);
             Sx[(int)round(kk)-1][(int)round(ll)-1] =
                MM[i][j];

             /* find max value of Sx so it can be
                normalized later */
                if(MM[i][j] > Sx_max) Sx_max = MM[i][j];
          }
      } // for j
   } // for i

   // Normalize Sx
      for(i=0; i<Np+1; i++)
      {
         for(j=0; j<2*N+1; j++)
         {
            Sx[i][j] = Sx[i][j]/Sx_max;
         }
      }

   // get fourth time hac (stop timing) and display:
      gettimeofday(&t1, NULL);
      i = timeval_subtract (&dummy_time2, &t1, &t0);
      cum_execution_time += dummy_time2.tv_sec + dummy_time2.tv_usec*1e-
6;

      printf("Total    Execution    time    (no    calls):    %3.6f\n",
cum_execution_time);
      printf("Total time spent doing FFTs (includes calls): %3.6f\n",
total_fft_time);
      printf("Time spent only doing the FFT: %3.6f\n", fft_only_time);
```

```
/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   PRINT OUTPUT FILE
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
   Output=fopen(Output_file, "w");
   if(Output == NULL)
   {
      puts("Error creating output file.");
      return(1);
   }

   for(i=0; i<Np+1; i++)
   {
      for(j=0; j<2*N+1; j++)
      {
         fprintf(Output, "%6.32e\t", Sx[i][j]);
      }
      fprintf(Output, "\n");
   }

   fclose(Output);

   printf("Results from FAM algorithm written to: %s\n",
      Output_file);

/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   FREE MEMORY AND EXIT
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
   free(fft1_in);   free(fft1_ot);
   free(fft1_scr1); free(fft1_scr2);
   free(fft2_in);   free(fft2_ot);
   free(fft2_scr1); free(fft2_scr2);
   printf("\nEnd of FAM Program Execution\n");

   return 0;
}
```

B. C SUBROUTINES

```
/* file name: complex_subrs.c */
#include <math.h>

struct complex
{
      double x; // real data
      double y; // imaginary data
};

/*struct complex complex_exp(double A);
struct complex complex_mult(struct complex A,
                            struct complex B);
```

```
struct complex complex_conj(struct complex A);
float complex_mag(struct complex A); */

struct complex complex_exp(double A)
{
   /* returns exp(j*A) = cos(A) + j*sin(A) */
      struct complex result;
      result.x = cos(A);
      result.y = sin(A);

      return(result);
}

struct complex complex_mult(struct complex A,
                            struct complex B)
{
    /*  FUNCTION DESCRIPTION
       This function multiplies two complex numbers
       in a+j*b form:
             (a + jb)(c + jd) = ac + jad + jbc + jjbd
                              = (ac - bd) + j(ad + bc)
    */
      struct complex result;
      result.x = (A.x * B.x) - (A.y * B.y);
      result.y = (A.x * B.y) + (A.y * B.x);

      return(result);
}

struct complex complex_conj(struct complex A)
{
      /*  FUNCTION DESCRIPTION
            This function returns the complex conjugate of A
      */
      struct complex result;
      result.x = A.x;
      result.y = - A.y;

      return(result);
}

float complex_mag(struct complex A)
{
    /*  FUNCTION DESCRIPTION
          This function returns the magnitude of the
      complex quantity A
      */

      float result;
      result = sqrt( (A.x*A.x) + (A.y*A.y) );

      return(result);
}
```

```
/* file name: other_subrs.c */
#include <math.h>

//float rem(float A, float B);

float rem(float A, float B)
{
   /*  FUNCTION DESCRIPTION
      rem returns the remainder of A / B
      note: this function assumes that both A and B
      are positive
   */
   float temp, result;
   temp = (int)(A/B);
   result = A - temp*B;

   return(result);
}
```

```
/* file name: fft.c */
#include <stdio.h>
#include <math.h>
#include <time.h>

#define PI 3.14159265358979

int bitrev(int a, int k);
int ilog2(int n);
void fft(double *bin, double *bout, double *a1,
         double *a2, int n);

/* bitrev(a, k) --
   reverse bits 0 thru k-1 in the integer "a"
*/
   int bitrev(int a, int k) {
      unsigned int i, b, p, q;
      for (i=b=0, p = 1, q = 1<<(k-1);
         i<k;
         i++, p <<= 1, q >>= 1 ) if (a & q) b |= p;
      return b;
}

/* ilog2(n) --
   return an integer log, base 2
*/
   int ilog2(int n) {
      int i;
      for (i=8*sizeof(int)-1;
         i>=0 && ((1<<i) & n)==0; i--);
      return i;
}

/* fft(b, a2, a1, n, sgn) -- do an n-point fft of complex
   vector b and return the result in b.  b consists of
   2*n FTYPE elements, organized as n complex pairs real_1,
```

```
   imag_1, real_2, imag_2, ..., real_n, imag_n. the arrays
   a1 and a2 are used for working storage and each has n
   FTYPE elements.  sgn is 1 for an FFT, and -1 for an
   IFFT.  The procedure is taken from the Cormen, Leiserson,
   and Rivest Algorithms text, in the section on efficient
   FFT implementations.  This implementation wastes some
   space; the b array should probably be dropped and the
   initial and final staging done in-place in the "a"
   arrays.
 */
void fft (double b[], double fft_out[], double a2[],
          double a1[], int n) {

   int i, j, k, k2, s, m, log2n;
   double wm1, wm2, w1, w2, t1, t2, u1, u2;
   int time0, time1, time2;

   log2n = ilog2(n);

   /* reorder input and split input into real and complex
      parts */
      for (i=0; i<n; i++)
      {
         j = bitrev(i,log2n);
         a1[j] = b[2*i];
         a2[j] = b[2*i+1];
      }

   /* loop on FFT stages */
      for (s=1; s<=log2n; s++)
      {
         m = 1<<s;   /* m = 2^s */
         wm1 = cos(2*PI/m); /* wm = exp(q*2*pi*i/m); */
         wm2 = sin(2*PI/m);

         w1 = 1.0;
         w2 = 0.0;

         for (j=0; j<m/2; j++)
         {
            for (k=j; k<n; k+=m)
            {
                  /* t = w*a[k+m/2]; */
               k2 = k+m/2;
               t1 = w1 * a1[k2]  -  w2 * a2[k2] ;
               t2 = w1 * a2[k2]  +  w2 * a1[k2] ;

               u1 = a1[k];
               u2 = a2[k];

               a1[k] = u1 + t1;
               a2[k] = u2 + t2;

               a1[k2] = u1 - t1;
               a2[k2] = u2 - t2;
```

```
            } // for k
            /* w = w * wm; */
            t1 = w1 * wm1  -  w2 * wm2 ;
            w2 = w1 * wm2  +  w2 * wm1 ;
            w1 = t1;
         } // for j
      } // for s

   /* flip the final stage */
      for (i=1; i<n/2; i++)
      {
         t1 = a1[i];
         a1[i] = a1[n-i];
         a1[n-i] = t1;
         t2 = a2[i];
         a2[i] = a2[n-i];
         a2[n-i] = t2;
      }

  /* copy out results */
      for (i=0; i<n; i++)
      {
         b[2*i] = a1[i];
         b[2*i+1] = a2[i];
      }

   for (i = 0; i<2*n; i++) fft_out[i] = b[i];
}
```

```
/* file name: high_prec_time.c */
int timeval_subtract (struct timeval *result,
                      struct timeval *x,
                      struct timeval *y);

/* Subtract the `struct timeval' values X and Y,
   storing the result in RESULT.  Return 1 if the
   difference is negative, otherwise 0. */

int timeval_subtract (result, x, y)
   struct timeval *result, *x, *y;
   {
      /* Perform the carry for the later subtraction
         by updating y. */
         if (x->tv_usec < y->tv_usec) {
         int nsec = (y->tv_usec - x->tv_usec) * 1e-6 + 1;
         y->tv_usec -= 1e6 * nsec;
         y->tv_sec += nsec;
   }

   if (x->tv_usec - y->tv_usec > 1e6)
   {
      int nsec = (x->tv_usec - y->tv_usec) * 1e-6;
      y->tv_usec += 1e6 * nsec;
      y->tv_sec -= nsec;
   }
```

```
  /* Compute the time remaining to wait.
     tv_usec is certainly positive. */
     result->tv_sec = x->tv_sec - y->tv_sec;
     result->tv_usec = x->tv_usec - y->tv_usec;

  /* Return 1 if result is negative. */
     return x->tv_sec < y->tv_sec;
}
```

APPENDIX E. SRC-6 SPECIFIC CODE FOR THE ALGORITHM

A. SRC MAIN C PROGRAM: GENERAL FFT ALGORITHM

```
#include <stdio.h>
#include <libmap.h>
#include <map.h>
#include <stdlib.h>
#include <time.h>
#include "high_prec_time.c"

#define pi 3.141592653589793

FILE *I_ptr; // pointer to the I-channel input file name

FILE *IFFT1_Out;  // pointer to the I-channel output file
                  // name for the first FFT results
FILE *QFFT1_Out;  // pointer to the Q-channel output file
                  // name for the first FFT results
FILE *IFFT2_Out;  // pointer to the I-channel output file
                  // name for the second FFT results
FILE *QFFT2_Out;  // pointer to the Q-channel output file
                  // name for the second FFT results

FILE *Output;     // pointer to the final data output file name

void channelize (double *, double (*)[], int64_t *, int64_t *,
                 int64_t *, int);
void fft_map (float *, float *, float *, int, int,
              int64_t *, int64_t *, int64_t *, int);
void downconvert (double (*)[], double (*)[], double (*)[],
                  double (*)[], double (*)[], double (*)[],
                  int64_t *, int64_t *, int64_t *, int);

int main()
{
   /* DECLARE VARIABLES AND CONSTANTS */
       /* declare file names and path */
         char I_file[] = "I_channel.txt";
         char Q_file[] = "Q_channel.txt";

         char IFFT1_Out_file[] = "IFFT1_out.txt";
         char QFFT1_Out_file[] = "QFFT1_out.txt";
         char IFFT2_Out_file[] = "IFFT2_out.txt";
         char QFFT2_Out_file[] = "QFFT2_out.txt";

         char Output_file[]    = "FAM_result.txt";

      /* Declare Input Variables */
         int fs = 7000; // sample frequency
         int df = 128;  // frequency resolution
         int M = 2;   // M = df/alpha
```

```
/* Declare all timing variables and get first time
   hac; start timing */
   struct timeval start1, start2, start3, temp_stop, time1;
   struct timeval subr_t0, subr_t1;
   float cum_time = 0.0, overall_time = 0.0;
   float channel_CALL_time, fft_CALL_time, downconvert_CALL_time;
   float channel_DMA_time = 0.0, channel_MAP_time = 0.0;
   float channel_channel_time = 0.0;
   float fft_DMA_time = 0.0, fft_MAP_time = 0.0;
   float fft_fft_time = 0.0;
   float downconvert_DMA_time = 0.0, downconvert_MAP_time = 0.0;
   float downconvert_downconvert_time = 0.0;
   int64_t map_time, t0, t1, t2;
   int timei;
   gettimeofday(&start1, NULL);

/* calculate dalpha */
   double dalpha = df/M;
/* determine number of input channels: fs/df */
   double Np = pow(2.0, ceil(log10(fs/df)/log10(2)) );
/* overlap factor in order to reduce the number of
   short time fft's.  L is the offset between points
   in the same column at consecutive rows. L shoud
   be less than or equal to Np/4
   (Prof. Loomis paper) */
   double L = Np/4;
/* determine number of columns formed in the
   channelization matrix (x) */
   double P = pow(2.0, ceil(log10(fs/dalpha/L)/log10(2)) );
/* determine total number of points in the input
   data to be processed */
   double N = P*L;

/* declare other variables and arrays to be used.
   Note: I tried to declare them in the order in which
   they are needed.  Some were consolidated.  */
   /* Loop Indexes */
      int i=0, j=0, k=0, index=0;
   /* Array to contain values from input file */
      float *I_Values;
      I_Values = (float*)malloc(N * sizeof(float));
   /* Initial Array and Matrix */
      double NN = (P-1)*L+Np; // resizes x
      double *x;
      x  = (double*)malloc(N * sizeof(double));
      int xInitialMax; // book-keeping on x array
   /* Declare Variables used for MAP Allocation */
      int nmap=1, mapnum=0;
   /* Declare Channelization Variables */
      double (*XW)[(int)P];
      XW = malloc(Np * P * sizeof(double));
   /* Declare FFT 1 Variables */
      /* Determine the number of rows needed
         fft1_N = number of points in first fft
```

```
            fft1_n = log2(N); or: 2^n = N             */
            float fft1_N, fft1_n = log10(Np)/log10(2);
            if (fft1_n <= 8)
            {
               fft1_N = 256;
               fft1_n = 8;
            }
            else if ((fft1_n > 8) & (fft1_n <= 14))
               {fft1_N = pow(2.0, fft1_n);}
            else
            {
               printf("The data size is too large for the ");
               printf("FFT algorithm to handle.\n");
               return(1);
            }
         double rad;  // will be used as a temp variable to
                      // generate twiddle table
         float *twiddle1, *fft1_in, *fft1_ot;
         twiddle1 = (float *)Cache_Aligned_Allocate(fft1_N *
            sizeof(float));
         fft1_in  = (float *)Cache_Aligned_Allocate(fft1_N *
            2 * sizeof(float));
         fft1_ot  = (float *)Cache_Aligned_Allocate(fft1_N *
            2 * sizeof(float));
         double (*Itemp_XF1)[(int)P], (*Qtemp_XF1)[(int)P];
         Itemp_XF1 = Cache_Aligned_Allocate(Np * P *
            sizeof(double));
         Qtemp_XF1 = Cache_Aligned_Allocate(Np * P *
            sizeof(double));
      /* Declare Downconversion Variables */
         double (*downtwiddleI)[(int)P], (*downtwiddleQ)[(int)P];
         downtwiddleI = Cache_Aligned_Allocate(Np * P *
            sizeof(double));
         downtwiddleQ = Cache_Aligned_Allocate(Np * P *
            sizeof(double));
         double (*IXF1)[(int)(Np*Np)], (*QXF1)[(int)(Np*Np)];
         IXF1 = Cache_Aligned_Allocate(P * Np * Np *
            sizeof(double));
         QXF1 = Cache_Aligned_Allocate(P * Np * Np *
            sizeof(double));
      /* Declare FFT 2 Variables */
         /* Determine the number of rows needed
            fft2_N = number of points in second fft
            fft2_n = log2(N); or: 2^n = N             */
            float fft2_N, fft2_n = log10(P)/log10(2);
            if (fft2_n <= 8)
            {
               fft2_N = 256;
               fft2_n = 8;
            }
            else if ((fft2_n > 8) & (fft2_n <= 14))
               {fft2_N = pow(2.0, fft2_n);}
            else
            {
               printf("The data size is too large for the ");
```

```
                printf("FFT algorithm to handle.\n");
                return(1);
             }
          float *twiddle2, *fft2_in, *fft2_ot;
          twiddle2 = (float *)Cache_Aligned_Allocate(fft2_N *
             sizeof(float));
          fft2_in  = (float *)Cache_Aligned_Allocate(fft2_N *
             2 * sizeof(float));
          fft2_ot  = (float *)Cache_Aligned_Allocate(fft2_N *
             2 * sizeof(float));

          double (*Itemp_XF2)[(int)(Np*Np)];
          double (*Qtemp_XF2)[(int)(Np*Np)];
          Itemp_XF2 = Cache_Aligned_Allocate(P * Np * Np *
             sizeof(double));
          Qtemp_XF2 = Cache_Aligned_Allocate(P * Np * Np *
             sizeof(double));
       /* Declare Final Output Variables */
         float IXF2[(int)P][(int)(Np*Np)], QXF2[(int)P][(int)(Np*Np)];
          double (*MM)[(int)(Np*Np)], (*Sx)[2*(int)N+1];
          float c, p, alpha, f, kk, ll, Sx_max;
          double Iscr, Qscr;
          int64_t joverNp, rem;
          Sx = Cache_Aligned_Allocate((Np+1) * ((2 * N)+1) *
             sizeof(double));
          MM = Cache_Aligned_Allocate( ( (int)((3*P/4)-(P/4))+1) *
             Np * Np * sizeof(double));

/* GET SECOND TIME HAC; STOP TIMING TO BRING DATA IN */
   gettimeofday(&temp_stop, NULL);
   timei = timeval_subtract (&time1, &temp_stop, &start1);
   cum_time = time1.tv_sec + time1.tv_usec*1.0e-6;
   overall_time = cum_time;

/* OPEN THE INPUT FILES */
   I_ptr = fopen(I_file, "r");
   if (I_ptr==NULL)
   {
      printf("Error opening I-channel input file.\n");
      return(1);
   }

/* READ IN THE I-CHANNEL FILE */
   /* This while loop reads in the first N values of the file
      and puts them into the I_Values array */
      while ( (fscanf(I_ptr, "%f", &I_Values[i]) != EOF) && (i<N) )
      {
         x[i] = I_Values[i];
         i++;
      }

   /* This loop fills the x array with zeros if there wasn't N
      rows of data in the input data file */
      if (i < N)
         while (i < N)
```

```
            {
               x[i] = 0;
               i++;
            }
      xInitialMax = i;
      fclose(I_ptr);

   /* GET THIRD TIME HAC; RESTART TIMING */
      gettimeofday(&start2, NULL);

/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   INPUT CHANNELIZATION - this part limits the total number of points
                          to be analyzed. It also generates a Np-by-P
                          matrix, X, with shifted versions of the input
                          vector in each column.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
   /* Zero fill x if we don't have NN samples.  The loop does the
      xx(NN) = 0 loop in the MATLAB code. */
      for(i=xInitialMax; i<NN;i++)  x[i] = 0;

   /* RESERVE MAP */
      if (map_allocate(nmap))
      {
         fprintf(stdout, "Map allocation failed for channelization.\n");
         exit(1);
      }

   /* Take time hac */
      gettimeofday(&temp_stop, NULL);
      timei = timeval_subtract (&time1, &temp_stop, &start2);
      cum_time += time1.tv_sec + time1.tv_usec*1.0e-6;
      gettimeofday(&subr_t0, NULL);

   /* Call Subroutine and Restart Timing */
      channelize(x, XW, &t0, &t1, &t2, mapnum);
      gettimeofday(&subr_t1, NULL);
      cum_time += t0*1e-8;
      channel_MAP_time      += t0*1e-8;
      channel_DMA_time      += t1*1e-8;
      channel_channel_time  += t2*1e-8;
      timei = timeval_subtract(&time1, &subr_t1, &subr_t0);
      channel_CALL_time += (time1.tv_sec + time1.tv_usec*1.0e-6) -
         t0*1e-8; // time to do execution - map time
      gettimeofday(&start3, NULL);

/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   FIRST FFT CALL
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
   /* USE THE cfft_fp32 ROUTINE. NOTE: MINIMUM POINT FFT IS 256
      SO I'LL NEED TO ADJUST IF I DON'T HAVE THAT MANY POINTS.
      I'LL INTERLEAVE ONE FFT INPUT VALUE WITH ZEROS BEHIND. */

   /* Generate Twiddle Table */
      i = 0;
      for(j=0; j< fft1_N/2; j++)
```

```
      {
         rad = 2.0*pi*((double)j/(double)fft1_N);
         twiddle1[i]   =  cos(rad);
         twiddle1[i+1] = -sin(rad);
         i += 2;
      }

   /* To do a 2-D FFT, do each column individually */
      for (i=0; i< P; i++)
      {
         /* Build FFT Input Matrix */
            k = 0;
            index = 0;
            for (j=0; j<2*fft1_N; j++)
            {
               if (j - (2*fft1_N/Np) * (int)(j*Np/(2*fft1_N) ) == 0)
               {
                  fft1_in[j] = XW[k][i];
                  k++;
               }
               else { fft1_in[j] = 0.0; }
            } // for j

         /* Take time hac */
            gettimeofday(&temp_stop, NULL);
            timei = timeval_subtract (&time1, &temp_stop, &start3);
            cum_time += time1.tv_sec + time1.tv_usec*1.0e-6;
            gettimeofday(&subr_t0, NULL);

         /* Call FFT and Restart Timing */
            fft_map(fft1_in, twiddle1, fft1_ot, fft1_n, 1,
                    &t0, &t1, &t2, 0);
            gettimeofday(&subr_t1, NULL);
            cum_time += t0*1e-8;
            fft_MAP_time  += t0*1e-8;
            fft_DMA_time  += t1*1e-8;
            fft_fft_time  += t2*1e-8;
            timei = timeval_subtract(&time1, &subr_t1, &subr_t0);
            fft_CALL_time += (time1.tv_sec + time1.tv_usec*1.0e-6) -
               t0*1e-8; // time to do execution - map time
            gettimeofday(&start3, NULL);

         /* Obtain FFT Output */
            k = 0;
            for (j=0; j<2*Np; j+=2)
            {
               Itemp_XF1[k][i] = fft1_N*fft1_ot[j + 0];
               Qtemp_XF1[k][i] = fft1_N*fft1_ot[j + 1];
               k++;
            }
      } // for i

   /* Since I go right back onto the map, I'll keep the one I have. */
   /* PRINT FFT 1 OUTPUT FILE
      IFFT1_Out=fopen(IFFT1_Out_file, "w");
```

```
      if(IFFT1_Out == NULL)
      {
         puts("Error creating FFT1 I-channel output file.");
         return(1);
      }

      QFFT1_Out=fopen(QFFT1_Out_file, "w");
      if(QFFT1_Out == NULL)
      {
         puts("Error creating FFT1 Q-channel output file.");
         return(1);
      }

      for(i=0; i<Np; i++)
      {
         for(j=0; j<P; j++)
         {
            fprintf(IFFT1_Out, "%6.32f\t", Itemp_XF1[i][j]);
            fprintf(QFFT1_Out, "%6.32f\t", Qtemp_XF1[i][j]);
         }
         fprintf(IFFT1_Out, "\n");
         fprintf(QFFT1_Out, "\n");
      }
      fclose(IFFT1_Out);
      fclose(QFFT1_Out);

      printf("I-Channel FFT1 Results from Cyclostationary FAM algorithm
written to: %s\n",
         IFFT1_Out_file);
      printf("Q-Channel FFT1 Results from Cyclostationary FAM algorithm
written to: %s\n",
         QFFT1_Out_file);
   */

/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   DOWNCONVERSION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
   /* Generate downconvert twiddle table */
      for (i=0; i<Np; i++)
      {
         k = i - ((int)Np/2);
         for (j=0; j<P; j++)
         {
            downtwiddleI[i][j] =  cos(2*pi*k*j*L/Np);
            downtwiddleQ[i][j] = -sin(2*pi*k*j*L/Np);
          }
       }

   /* Take time hac */
      gettimeofday(&temp_stop, NULL);
      timei = timeval_subtract (&time1, &temp_stop, &start3);
      cum_time += time1.tv_sec + time1.tv_usec*1.0e-6;
      gettimeofday(&subr_t0, NULL);

   /* Call Subroutine and Restart Timing */
```

```
      downconvert(Itemp_XF1, Qtemp_XF1,
                  downtwiddleI, downtwiddleQ,
                  IXF1, QXF1, &t0, &t1, &t2, mapnum);
      gettimeofday(&subr_t1, NULL);
      cum_time += t0*1e-8;
      downconvert_MAP_time          += t0*1e-8;
      downconvert_DMA_time          += t1*1e-8;
      downconvert_downconvert_time  += t2*1e-8;
      timei = timeval_subtract(&time1, &subr_t1, &subr_t0);
      downconvert_CALL_time += (time1.tv_sec + time1.tv_usec*1.0e-6) -
         t0*1e-8; // time to do execution - map time
      gettimeofday(&start3, NULL);

   /* Since I go right back onto the map, I'll keep the one I have. */

   /* The following loop was used to generate data for G. Upperman's
      Thesis */
      printf("********* THESIS DATA: Downconversion *****\n");
      for (i=0; i<5; i++)
      {
         printf("\t%2.8f+i*%2.8f\n", IXF1[i][1], QXF1[i][1]);
      }

/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   SECOND FFT CALL
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
   /* USE THE cfft_fp32 ROUTINE. NOTE: MINIMUM POINT FFT IS 256
      SO I'LL NEED TO ADJUST IF I DON'T HAVE THAT MANY POINTS.
      I'LL INTERLEAVE ONE FFT INPUT VALUE WITH ZEROS BEHIND. */
   /* Note: Use different twiddle table then before
      i = 0;
      for(j=0; j< fft2_N/2; j++)
      {
         rad = 2.0*pi*((double)j/(double)fft2_N);
         twiddle2[i]   =  cos(rad);
         twiddle2[i+1] = -sin(rad);
         i += 2;
      }

   /* To do a 2-D FFT, do each column individually */
      for (i=0; i< Np*Np; i++) // col
      {
         /* Build FFT Input Matrix */
            k = 0;
            index = 0;
            for (j=0; j<2*fft2_N; j++)
            {
               if (j - (2*fft2_N/P) * (int)( j*P/(2*fft2_N) ) == 0)
               {
                  fft2_in[j + 0] = IXF1[k][i];
                  fft2_in[j + 1] = QXF1[k][i];
                  k++;
               }
               else { fft2_in[j] = 0.0; }
            } // for j
```

```
      /* Take time hac */
         gettimeofday(&temp_stop, NULL);
         timei = timeval_subtract (&time1, &temp_stop, &start3);
         cum_time += time1.tv_sec + time1.tv_usec*1.0e-6;
         gettimeofday(&subr_t0, NULL);

      /* Call FFT and Restart Timing */
         fft_map(fft2_in, twiddle2, fft2_ot, fft2_n, 1, &t0,
            &t1, &t2, 0);
         gettimeofday(&subr_t1, NULL);
         cum_time += t0*1e-8;
         fft_MAP_time  += t0*1e-8; // time of map routine
         fft_DMA_time  += t1*1e-8; // time of DMA transfers
         fft_fft_time  += t2*1e-8; // time of fft algorithm
         timei = timeval_subtract(&time1, &subr_t1, &subr_t0);
         fft_CALL_time += (time1.tv_sec + time1.tv_usec*1.0e-6) -
            t0*1e-8; // time to do execution - map time
         gettimeofday(&start3, NULL);

      /* In Debug mode, the FFT 2 loop takes a LONG time (minutes)
         to complete.  Use this code to keep track of status
         if (i - (200.0) * ((int)(i/200.0)) == 0)
         {
            printf("Second FFT: %3.2f%% complete.\n",
               i*100/(Np*Np) );
         }
      */

      /* Get FFT Output */
         k = 0;
         for (j=0; j<2*P; j+=2)
         {
            Itemp_XF2[k][i] = fft2_N*fft2_ot[j + 0];
            Qtemp_XF2[k][i] = fft2_N*fft2_ot[j + 1];
            k++;
         }
   } // for i

/* free map */
   if (map_free (nmap))
   {
      printf("Map deallocation failed for downconversion.\n");
      exit(1);
   }

/* PRINT FILE
   IFFT2_Out=fopen(IFFT2_Out_file, "w");
   if(IFFT2_Out == NULL)
   {
      puts("Error creating FFT2 I-channel output file.");
      return(1);
   }

   QFFT2_Out=fopen(QFFT2_Out_file, "w");
```

```
      if(QFFT2_Out == NULL)
      {
         puts("Error creating FFT2 Q-channel output file.");
         return(1);
      }

      for(i=0; i<P; i++)
      {
         for(j=0; j<Np*Np; j++)
         {
            fprintf(IFFT2_Out, "%6.32f\t", Itemp_XF2[i][j]);
            fprintf(QFFT2_Out, "%6.32f\t", Qtemp_XF2[i][j]);
         }
         fprintf(IFFT2_Out, "\n");
         fprintf(QFFT2_Out, "\n");
      }
      fclose(IFFT2_Out);
      fclose(QFFT2_Out);

      printf("I-Channel FFT2 Results from Cyclostationary FAM algorithm
written to: %s\n",
         IFFT2_Out_file);
      printf("Q-Channel FFT2 Results from Cyclostationary FAM algorithm
written to: %s\n",
         QFFT2_Out_file);
   */

/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   FINAL MATRIX MANIPULATION AND OUTPUT
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
        /* //Call loop ;
           // Take time hac
         gettimeofday(&temp_stop, NULL);
           timei = timeval_subtract (&time1, &temp_stop, &start3);
          cum_time += time1.tv_sec + time1.tv_usec*1.0e-6;

           outprep(Itemp_XF2, Qtemp_XF2, MM, &map_time, mapnum);

           cum_time += map_time*1e-8;
         gettimeofday(&start3, NULL);

        // free map
          if (map_free (nmap)) {
             printf("Map deallocation failed for downconversion.\n");
             exit(1);
          }

*/



/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   MATRIX MANIPULATION - implements the FFT shift and left/right flip
                         in the matlab code in one single loop. End
                         result is that the top and bottom halves of
```

```
                          the FFT are swapped.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
   for(i=0; i<=(P/2-1); i++)
   {
      for(j=0; j<Np*Np; j++)
      {
         /* Bottom real half becomes top real half */
            IXF2[i][j] = Itemp_XF2[i+(int)(P/2)][j];
         /* Top real half becomes bottom real half */
            IXF2[i + (int)(P/2)][j] = Itemp_XF2[i][j];


         /* Bottom imag half becomes top imag half */
            QXF2[i][j] = Qtemp_XF2[i+(int)(P/2)][j];
         /* Top imag half becomes bottom imag half */
            QXF2[i + (int)(P/2)][j] = Qtemp_XF2[i][j];
      } // for j
   } // for i

   /* Obtain the magnitude of the complex values */
      Sx_max = 0;
      for(i=P/4-1; i<(3*P/4); i++)
      {
         for(j=0; j<Np*Np; j++)
         {
            /* Temp Scratch Variables */
               Iscr = IXF2[i][j];
               Qscr = QXF2[i][j];
            /* Take Magnitude */
               MM[i-(int)(P/4)+1][j] = sqrt((Iscr*Iscr) + (Qscr*Qscr));
            /* Keep track of the maximum value; will be used later
               to normalize the final result
               if(MM[i-(int)(P/4)+1][j] > Sx_max)
                  Sx_max = MM[i-(int)(P/4)+1][j];*/
         }
      }

/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   DATA DISPLAY - display only the data inside the range of interest -
                  centralizes the bi-frequency plane according to
                  alpha0 and f0 vectors.  Note: the alpha0 and f0
                  vectors are defined as follows (in matlab terms):
                     alpha0 = -fs  :fs/N :fs;
                     f0     = -fs/2:fs/Np:fs/2;
                  but are not declared in this program since they
                  are only used for plotting the results.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
   /* Clear Sx matrix since not every location is necessarily
      written to.  Seems like this loop is unnecessary, but
      I had instances where old data in the memory was being used */
      for (i = 0; i<Np+1; i++)
      {
         for (j=0; j<2*N+1; j++)
         {
            Sx[i][j] = 0;
```

```
         }
      }

   /* Determine Final Output Order */
      for(i=0; i<=P/2; i++)
      {
         for(j=0; j<Np*Np; j++)
         {
            joverNp = (int)((j+1)/Np);
            rem = (j+1) - Np*joverNp;

            if(rem == 0)
            {
               c = Np/2 - 1;
            }
            else
            {
               c = rem - Np/2 - 1;
            }

            k = joverNp - Np/2;
            p = i - P/4;

            alpha = ((k-c)/Np) + ((p-1)/N);
            f = (k+c)/(2*Np);

            if (((alpha > -1) & (alpha < 1)) | ((f >-0.5) & (f < 0.5)))
            {
               kk = 1+Np*(f + .5);
               if ( (kk-(int)kk) < 0.5) kk = (int)kk;
               else kk = (int)kk + 1;

               ll = 1+N*(alpha + 1);
               if ( (ll-(int)ll) < 0.5) ll = (int)ll;
               else ll = (int)ll + 1;

               Sx[(int)kk-1][(int)ll-1] = MM[i][j];

            /* Keep track of the maximum value; will be used later
               to normalize the final result */
               if(MM[i][j] > Sx_max)
                  Sx_max = MM[i][j];
            } // if
         } // for j
      } // for i


   /* Normalize Sx - ORIGINAL */
      for(i=0; i<Np+1; i++)
      {
         for(j=0; j<2*N+1; j++)
         {
            Sx[i][j] = Sx[i][j]/Sx_max;
         }
      }
```

```
   /* get fourth time hac (stop timing) and display: */
      gettimeofday(&time1, NULL);
      timei = timeval_subtract (&time1, &temp_stop, &start3);
      cum_time += time1.tv_sec + time1.tv_usec*1.0e-6;
      timei = timeval_subtract (&time1, &temp_stop, &start2);
      overall_time += time1.tv_sec + time1.tv_usec*1.0e-6;
      printf("Execution times:\n");
      printf("   %3.6f seconds total\n", overall_time);
      printf("Of the total time:\n");
      printf("   %3.6f seconds were spent on the CALLS to FFTs\n",
         fft_CALL_time);
      printf("   %3.6f seconds were spent on the MAP for FFTs\n",
         fft_MAP_time);
      printf("   Of the time spent on the MAP for FFTs:\n");
      printf("      %3.6f seconds were spent on DMAs\n",
         fft_DMA_time);
      printf("      %3.6f seconds were spent in the FFT loop\n",
         fft_fft_time);

      printf("   %3.6f seconds were spent on the CALLS to Channelize\n",
         channel_CALL_time);
      printf("   %3.6f seconds were spent on the MAP for Channelize\n",
         channel_MAP_time);
      printf("   Of the time spent on the MAP for Channelize:\n");
      printf("      %3.6f seconds were spent on DMAs\n",
         channel_DMA_time);
      printf("      %3.6f seconds were spent channelizing\n",
         channel_channel_time);

      printf("       %3.6f   seconds   were   spent   on   the   CALLS   to
Downconvert\n",
         downconvert_CALL_time);
      printf("   %3.6f seconds were spent on the MAP for Downconvert\n",
         downconvert_MAP_time);
      printf("   Of the time spent on the MAP for Downconvert:\n");
      printf("      %3.6f seconds were spent on DMAs\n",
         downconvert_DMA_time);
      printf("      %3.6f seconds were spent downconverting\n",
         downconvert_downconvert_time);
      printf("Execution  time  not  including  calls  and  data  transfers:
%3.6f seconds\n",
         overall_time - fft_CALL_time - fft_DMA_time - channel_CALL_time
-
         channel_DMA_time        -        downconvert_CALL_time        -
downconvert_DMA_time);

   /* PRINT OUTPUT FILE */
      Output=fopen(Output_file, "w");
      if(Output == NULL)
      {
         puts("Error creating output file.");
         return(1);
      }
```

```
      for(i=0; i<Np + 1; i++)
      {
         for(j=0; j<2*N + 1; j++)
         {
            fprintf(Output, "%6.32f\t", Sx[i][j]);
         }
         fprintf(Output, "\n");
      }
      fclose(Output);

      printf("\nResults from Cyclostationary FAM algorithm written to:
%s\n", Output_file);

   printf("\nEnd of FAM Program Execution\n");
   return 0;
}
```

B. SRC MAIN C PROGRAM: CUSTOM FFT ALGORITHM

```
#include <stdio.h>
#include <libmap.h>
#include <map.h>
#include <stdlib.h>
#include <time.h>
#include "high_prec_time.c"

#define pi 3.141592653589793

FILE *I_ptr; // pointer to the I-channel input file name

FILE *IFFT1_Out;  // pointer to the I-channel output file name
                  // for the first FFT results
FILE *QFFT1_Out;  // pointer to the Q-channel output file name
                  // for the first FFT results
FILE *IFFT2_Out;  // pointer to the I-channel output file name
                  // for the second FFT results
FILE *QFFT2_Out;  // pointer to the Q-channel output file name
                  // for the second FFT results

FILE *Output;     // pointer to the final data output file name

void channelize (double *, double (*)[], int64_t *, int64_t *,
                 int64_t *, int);

void fft (double *Iin, double *Qin, double *Iot, double *Qot, int n,
          int64_t *, int64_t *, int64_t *, int map);

void downconvert (double (*)[], double (*)[], double (*)[],
                  double (*)[], double (*)[], double (*)[],
                  int64_t *, int64_t *, int64_t *, int);

int main()
{
   /* DECLARE VARIABLES AND CONSTANTS */
      /* declare file names and path */
```

```
         char I_file[] = "I_channel.txt";

         char IFFT1_Out_file[] = "IFFT1_out.txt";
         char QFFT1_Out_file[] = "QFFT1_out.txt";
         char IFFT2_Out_file[] = "IFFT2_out.txt";
         char QFFT2_Out_file[] = "QFFT2_out.txt";

         char Output_file[] = "FAM_result.txt";

      /* Declare Input Variables */
         int fs = 7000; // sample frequency
         int df = 128;  // frequency resolution
         int M = 2;   // M = df/alpha

      /* Declare all timing variables and get first time hac;
         start timing */
         struct timeval start1, start2, start3, temp_stop, time1;
         struct timeval subr_t0, subr_t1;
         float cum_time = 0.0, overall_time = 0.0;
         float channel_CALL_time, fft_CALL_time, downconvert_CALL_time;
         float channel_DMA_time = 0.0, channel_MAP_time = 0.0;
         float channel_channel_time = 0.0;
         float fft_DMA_time = 0.0, fft_MAP_time = 0.0;
         float fft_fft_time = 0.0;
         float downconvert_DMA_time = 0.0, downconvert_MAP_time = 0.0;
         float downconvert_downconvert_time = 0.0;
         int64_t map_time, t0, t1, t2;
         int timei;
         gettimeofday(&start1, NULL);

      /* calculate dalpha */
         double dalpha = df/M;
      /* determine number of input channels: fs/df */
         double Np = pow(2.0, ceil(log10(fs/df)/log10(2)) );
      /* overlap factor in order to reduce the number of short time
         fft's. L is the offset between points in the same column at
         consecutive rows.  L shoud be less than or equal to Np/4
         (Prof. Loomis paper) */
         double L = Np/4;
      /* determine number of columns formed in the
         channelization matrix (x) */
         double P = pow(2.0, ceil(log10(fs/dalpha/L)/log10(2)) );
      /* determine total number of points in the input data to
         be processed */
         double N = P*L;

      /* declare other variables and arrays to be used
         Note: I tried to declare them in the order in which they
         are needed.  Some were consolidated. */
         /* Loop Indexes */
            int i = 0, j = 0, k=0, index = 0;
         /* Array to contain values from input file */
            float *I_Values;
            I_Values = (float*)malloc(N * sizeof(float));
         /* Initial Array and Matrix */
```

```
            double NN = (P-1)*L+Np; // resizes x array
            double *x;
            x  = (double*)malloc(NN * sizeof(double));
            int xInitialMax; // book-keeping on x array
         /* Declare Variables used for MAP allocation */
            int nmap = 1, mapnum = 0;
         /* Declare Channelization Variables */
            double (*XW)[(int)P];
            XW = malloc(Np * P * sizeof(double));
         /* Declare FFT 1 Variables */
            double *fft1_in1, *fft1_in2, *fft1_ot1, *fft1_ot2;
            fft1_in1  = (double *)Cache_Aligned_Allocate(Np *
               sizeof(double));
            fft1_in2  = (double *)Cache_Aligned_Allocate(Np *
               sizeof(double));
            fft1_ot1  = (double *)Cache_Aligned_Allocate(Np *
               sizeof(double));
            fft1_ot2  = (double *)Cache_Aligned_Allocate(Np *
               sizeof(double));

            double (*Itemp_XF1)[(int)P], (*Qtemp_XF1)[(int)P];
            Itemp_XF1 = Cache_Aligned_Allocate(Np * P * sizeof(double));
            Qtemp_XF1 = Cache_Aligned_Allocate(Np * P * sizeof(double));

         /* Declare Downconversion Variables */
            double (*downtwiddleI)[(int)P], (*downtwiddleQ)[(int)P];
            downtwiddleI = Cache_Aligned_Allocate(Np * P *
               sizeof(double));
            downtwiddleQ = Cache_Aligned_Allocate(Np * P *
               sizeof(double));
            double (*IXF1)[(int)(Np*Np)], (*QXF1)[(int)(Np*Np)];
            IXF1 = Cache_Aligned_Allocate(Np * Np * P *
               sizeof(double));
            QXF1 = Cache_Aligned_Allocate(Np * Np * P *
               sizeof(double));

         /* Declare FFT 2 Variables */
            double *fft2_in1, *fft2_in2, *fft2_ot1, *fft2_ot2;
            fft2_in1  = (double *)Cache_Aligned_Allocate(P *
               sizeof(double));
            fft2_in2  = (double *)Cache_Aligned_Allocate(P *
               sizeof(double));
            fft2_ot1  = (double *)Cache_Aligned_Allocate(P *
               sizeof(double));
            fft2_ot2  = (double *)Cache_Aligned_Allocate(P *
               sizeof(double));
            double (*Itemp_XF2)[(int)(Np*Np)];
            double (*Qtemp_XF2)[(int)(Np*Np)];
            Itemp_XF2 = Cache_Aligned_Allocate(P * Np * Np *
               sizeof(double));
            Qtemp_XF2 = Cache_Aligned_Allocate(P * Np * Np *
               sizeof(double));

         /* Declare Output Variables */
           float IXF2[(int)P][(int)(Np*Np)], QXF2[(int)P][(int)(Np*Np)];
```

```
            double (*MM)[(int)(Np*Np)], (*Sx)[2*(int)N+1];
            float c, p, alpha, f, kk, ll, Sx_max;
            double Iscr, Qscr;
            int64_t joverNp, rem;
            Sx = Cache_Aligned_Allocate((Np+1) * ((2 * N)+1) *
               sizeof(double));
            MM = Cache_Aligned_Allocate( ( (int)((3*P/4)-(P/4))+1) *
               Np * Np * sizeof(double));

   /* GET SECOND TIME HAC; STOP TIMING TO BRING DATA IN */
      gettimeofday(&temp_stop, NULL);
      timei = timeval_subtract (&time1, &temp_stop, &start1);
      cum_time = time1.tv_sec + time1.tv_usec*1.0e-6;
      overall_time = cum_time;

   /* OPEN THE INPUT FILES */
      I_ptr = fopen(I_file, "r");
      if (I_ptr==NULL)
      {
         printf("Error opening I-channel input file.\n");
         return(1);
      }

   /* READ IN THE I-CHANNEL FILE */
      while ( (fscanf(I_ptr, "%f", &I_Values[i]) != EOF) && (i<N) )
      {
         x[i] = I_Values[i];
         i++;
      }

      /* This Loop fills the x array with zeros if there wasn't N
         rows of data in the input file */
         if (i < N)
            while (i < N)
            {
               x[i] = 0;
               i++;
            }
         xInitialMax = i;
         fclose(I_ptr);

   /* GET THIRD TIME HAC; RESTART TIMING */
      gettimeofday(&start2, NULL);

/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   INPUT CHANNELIZATION - this part limits the total number of points to
          be analyzed. It also generates a Np-by-P matrix, X, with
          shifted versions of the input vector in each column.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
   /* Zero fill x if we don't have NN samples.  The loop does the
      xx(NN) = 0 loop in the Matlab code. */
      for(i=xInitialMax; i<NN;i++)  x[i] = 0;

   /* Reserve MAP */
      if (map_allocate(nmap))
```

```
      {
         fprintf(stdout, "Map allocation failed for channelization.\n");
         exit(1);
      }

   /* Take time hac */
      gettimeofday(&temp_stop, NULL);
      timei = timeval_subtract (&time1, &temp_stop, &start2);
      cum_time += time1.tv_sec + time1.tv_usec*1.0e-6;
      gettimeofday(&subr_t0, NULL);

   /* Call subroutine and Restart Timing */
      channelize(x, XW, &t0, &t1, &t2, mapnum);
      gettimeofday(&subr_t1, NULL);
      cum_time += t0*1e-8;
      channel_MAP_time      += t0*1e-8;
      channel_DMA_time      += t1*1e-8;
      channel_channel_time  += t2*1e-8;
      timei = timeval_subtract(&time1, &subr_t1, &subr_t0);
      channel_CALL_time += (time1.tv_sec + time1.tv_usec*1.0e-6) -
         t0*1e-8; // time to do execution - map time
      gettimeofday(&start3, NULL);

/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   FIRST FFT CALL - To do a 2-D FFT, do each column individually
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
   for (i=0; i< P; i++)
   {
      /* Build FFT Input Matrix */
         for (j=0; j<Np; j++)
         {
            fft1_in1[j] = XW[j][i];
            fft1_in2[j] = 0;
          } // for j

      /* Take time hac */
         gettimeofday(&temp_stop, NULL);
         timei = timeval_subtract (&time1, &temp_stop, &start3);
         cum_time += time1.tv_sec + time1.tv_usec*1.0e-6;
         gettimeofday(&subr_t0, NULL);

      /* Call FFT and restart timing */
         fft(fft1_in1, fft1_in2, fft1_ot1, fft1_ot2, Np,
            &t0, &t1, &t2, 0);
         gettimeofday(&subr_t1, NULL);
         cum_time += t0*1e-8;
         fft_MAP_time  += t0*1e-8;
         fft_DMA_time  += t1*1e-8;
         fft_fft_time  += t2*1e-8;
         timei = timeval_subtract(&time1, &subr_t1, &subr_t0);
         fft_CALL_time += (time1.tv_sec + time1.tv_usec*1.0e-6) -
            t0*1e-8; // time to do execution - map time
         gettimeofday(&start3, NULL);

      /* Obtain FFT Output */
```

```
          for (j=0; j<Np; j++)
          {
             Itemp_XF1[j][i] = fft1_ot1[j];
             Qtemp_XF1[j][i] = fft1_ot2[j];
             }
          }   // for i

       /* Since, I go right back onto the MAP, I'll keep the one I have
          allocated */

       /* PRINT FILE
          IFFT1_Out=fopen(IFFT1_Out_file, "w");
          if(IFFT1_Out == NULL)
          {
             puts("Error creating FFT1 I-channel output file.");
             return(1);
          }

          QFFT1_Out=fopen(QFFT1_Out_file, "w");
          if(QFFT1_Out == NULL)
          {
             puts("Error creating FFT1 Q-channel output file.");
             return(1);
          }

          for(i=0; i<Np; i++)
          {
             for(j=0; j<P; j++)
             {
                fprintf(IFFT1_Out, "%6.32e\t", Itemp_XF1[i][j]);
                fprintf(QFFT1_Out, "%6.32e\t", Qtemp_XF1[i][j]);
             }
             fprintf(IFFT1_Out, "\n");
             fprintf(QFFT1_Out, "\n");
          }
          fclose(IFFT1_Out);
          fclose(QFFT1_Out);

          printf("I-Channel FFT1 Results from ");
          printf("Cyclostationary FAM algorithm written to: %s\n",
             IFFT1_Out_file);
          printf("Q-Channel FFT1 Results from ");
          printf("Cyclostationary FAM algorithm written to: %s\n",
             QFFT1_Out_file);
*/

/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   DOWNCONVERSION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
   /* Reserve MAP
      if (map_allocate(nmap))
      {
         fprintf(stdout, "Map allocation failed for channelization.\n");
         exit(1);
       }*/
```

```
   /* Generate downconvert twiddle table */
      for (i=0; i<Np; i++)
      {
         k = i - ((int)Np/2);

         for (j=0; j<P; j++)
         {
            downtwiddleI[i][j] =  cos(-2*pi*k*j*L/Np);
            downtwiddleQ[i][j] = sin(-2*pi*k*j*L/Np);
         }
      }

   /* Take time hac */
      gettimeofday(&temp_stop, NULL);
      timei = timeval_subtract (&time1, &temp_stop, &start3);
      cum_time += time1.tv_sec + time1.tv_usec*1.0e-6;
      gettimeofday(&subr_t0, NULL);

   /* Call loop */;
      downconvert(Itemp_XF1, Qtemp_XF1,
                  downtwiddleI, downtwiddleQ,
                  IXF1, QXF1, &t0, &t1, &t2, mapnum);
      gettimeofday(&subr_t1, NULL);
      cum_time += t0*1e-8;
      downconvert_MAP_time          += t0*1e-8;
      downconvert_DMA_time          += t1*1e-8;
      downconvert_downconvert_time  += t2*1e-8;
      timei = timeval_subtract(&time1, &subr_t1, &subr_t0);
      downconvert_CALL_time += (time1.tv_sec + time1.tv_usec*1.0e-6) -
         t0*1e-8; // time to do execution - map time
      gettimeofday(&start3, NULL);

   /* free map
      if (map_free (nmap))
      {
         printf("Map deallocation failed for channelization.\n");
         exit(1);
      }  */

   /* The following nested loop was used to generate data for G.
      Upperman's Thesis */
      printf("********* THESIS DATA: Downconversion *****\n");
      for (i=0; i<5; i++)
      {
         printf("\t%2.8f+i*%2.8f\n", IXF1[i][1], QXF1[i][1]);
      }

/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   SECOND FFT CALL
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
   /* Reserve MAP
      if (map_allocate(1))
      {
         fprintf(stdout, "Map allocation failed for FFT 1.\n");
         exit(1);
```

```
      }*/

   for (i=0; i< Np*Np; i++) // col
   {
      /* Build FFT Input matrix */
         for (j=0; j<P; j++)
         {
            fft2_in1[j] = IXF1[j][i];
            fft2_in2[j] = QXF1[j][i];
         } // for j

      /* Take time hac */
         gettimeofday(&temp_stop, NULL);
         timei = timeval_subtract (&time1, &temp_stop, &start3);
         cum_time += time1.tv_sec + time1.tv_usec*1.0e-6;
         gettimeofday(&subr_t0, NULL);

      /* Call fft and Restart Timing */
         fft(fft2_in1, fft2_in2, fft2_ot1, fft2_ot2, 8,
            &t0, &t1, &t2, 0);
         gettimeofday(&subr_t1, NULL);
         cum_time += t0*1e-8;
         fft_MAP_time  += t0*1e-8; // time of map routine
         fft_DMA_time  += t1*1e-8; // time of DMA transfers
         fft_fft_time  += t2*1e-8; // time of fft algorithm
         timei = timeval_subtract(&time1, &subr_t1, &subr_t0);
         fft_CALL_time += (time1.tv_sec + time1.tv_usec*1.0e-6) -
            t0*1e-8; // time to do execution - map time
         gettimeofday(&start3, NULL);

      /* In Debug mode, the FFT 2 loop takes a LONG time (minutes)
         to complete.  Use this code to keep track of status
         if (i - (200.0) * ((int)(i/200.0)) == 0)
         {
            printf("Second FFT: %3.2f%% complete.\n", i*100/(Np*Np) );
         }*/

      /* Obtain FFT Output */
         for (j=0; j<P; j++)
         {
            Itemp_XF2[j][i] = fft2_ot1[j];
            Qtemp_XF2[j][i] = fft2_ot2[j];
         }

   } // for i

   /* Free map */
      if (map_free (nmap))
      {
         printf("Map deallocation failed for downconversion.\n");
         exit(1);
      }

   /* PRINT FILE
      IFFT2_Out=fopen(IFFT2_Out_file, "w");
```

```
      if(IFFT2_Out == NULL)
      {
         puts("Error creating FFT2 I-channel output file.");
         return(1);
      }

      QFFT2_Out=fopen(QFFT2_Out_file, "w");
      if(QFFT2_Out == NULL)
      {
         puts("Error creating FFT2 Q-channel output file.");
         return(1);
      }

      for(i=0; i<P; i++)
      {
         for(j=0; j<Np*Np; j++)
         {
            fprintf(IFFT2_Out, "%6.32f\t", Itemp_XF2[i][j]);
            fprintf(QFFT2_Out, "%6.32f\t", Qtemp_XF2[i][j]);
         }
         fprintf(IFFT2_Out, "\n");
         fprintf(QFFT2_Out, "\n");
      }
      fclose(IFFT2_Out);
      fclose(QFFT2_Out);

      printf("I-Channel FFT2 Results from Cyclostationary FAM algorithm
written to: %s\n", IFFT2_Out_file);
      printf("Q-Channel FFT2 Results from Cyclostationary FAM algorithm
written to: %s\n", QFFT2_Out_file);
   */

/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   FINAL MATRIX MANIPULATION AND OUTPUT
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
   /* Call Subroutine */
      //   outprep(Itemp_XF2, Qtemp_XF2, Sx, mapnum);

   /* Free map
      if (map_free (nmap))
      {
         printf("Map deallocation failed for downconversion.\n");
         exit(1);
      }    */

/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   MATRIX MANIPULATION - implements the FFT shift and left/right flip in
          the matlab code in one single loop
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
   /* Swap bottom and top halves: */
      for(i=0; i<=(P/2-1); i++)
      {
         for(j=0; j<Np*Np; j++)
         {
            /* Bottom real half becomes top real half */
```

```
            IXF2[i][j] = Itemp_XF2[i+(int)(P/2)][j];
         /* Top real half becomes bottom real half */
            IXF2[i + (int)(P/2)][j] = Itemp_XF2[i][j];

         /* Bottom imaginary half becomes top imaginary half */
            QXF2[i][j] = Qtemp_XF2[i+(int)(P/2)][j];
         /* Top imaginary half becomes bottom imaginary half */
            QXF2[i + (int)(P/2)][j] = Qtemp_XF2[i][j];
      } // for j
   } // for i

/* Obtain the magnitude of the complex values */
   for(i=(P/4)-1; i<(3*P/4); i++)
   {
      for(j=0; j<Np*Np; j++)
      {
         Iscr = IXF2[i][j];
         Qscr = QXF2[i][j];
         MM[i-(int)(P/4)+1][j] = sqrtf( (Iscr*Iscr) + (Qscr*Qscr) );
      } // for j
   } // for i

/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   DATA DISPLAY - display only the data inside the range of interest -
          centralizes the bi-frequency plane according to alpha0 and f0
          vectors.  Note: the alpha0 and f0 vectors are defined as
          follows (in matlab terms):
             alpha0 = -fs  :fs/N :fs;
             f0     = -fs/2:fs/Np:fs/2;
          but are not declared in this program since they are only used
          for plotting the results.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
   Sx_max = 0;

   /* Clear Sx matrix since not every location is necessarily written
to.
      Seems like this loop is unnecessary, but I had instances where old
      data in the memory was being used. */
      for (i = 0; i<Np+1; i++)
      {
         for (j=0; j<2*N+1; j++)
         {
            Sx[i][j] = 0;
         } // for j
      } // for i

   /* Determine Final Output */
      for(i=0; i<=.5*P; i++)
      {
         for(j=0; j<Np*Np; j++)
         {
            joverNp = (int)((j+1)/Np);
            rem = (j+1) - Np*joverNp;

            if(rem == 0)
```

```
            {
               c = .5*Np - 1;
            }
            else
            {
               c = rem - .5*Np - 1;
            }

            k = joverNp - .5*Np;
            p = i - .25*P;

            alpha = ((k-c)/Np) + ((p-1)/N);
            f = .5*(k+c)/Np;

            if (((alpha > -1) & (alpha < 1)) | ((f >-.5) & (f < .5)))
            {
               kk = 1+Np*(f + .5);
               if ( (kk-(int)kk) < .5) kk = (int)kk;
               else kk = (int)kk + 1;

               ll = 1+N*(alpha + 1);
               if ( (ll-(int)ll) < .5) ll = (int)ll;
               else ll = (int)ll + 1;

               Sx[(int)kk-1][(int)ll-1] = MM[i][j];

               /* find max value of Sx so it can be normalized later */
                  if(MM[i][j] > Sx_max) {Sx_max = MM[i][j];}
            } // end if
         } // for j
      } // for i

   /* Normalize Sx - ORIGINAL */
      for(i=0; i<Np+1; i++)
      {
         for(j=0; j<2*N+1; j++)
         {
            Sx[i][j] = Sx[i][j]/Sx_max;
         }
      }

   /* get fourth time hac (stop timing) and display: */
      gettimeofday(&time1, NULL);
      timei = timeval_subtract (&time1, &temp_stop, &start3);
      cum_time += time1.tv_sec + time1.tv_usec*1.0e-6;
      timei = timeval_subtract (&time1, &temp_stop, &start2);
      overall_time += time1.tv_sec + time1.tv_usec*1.0e-6;
      printf("Execution times:\n");
      printf("   %3.6f seconds total\n", overall_time);
      printf("Of the total time:\n");
      printf("   %3.6f seconds were spent on the CALLS to FFTs\n",
         fft_CALL_time);
      printf("   %3.6f seconds were spent on the MAP for FFTs\n",
         fft_MAP_time);
      printf("   Of the time spent on the MAP for FFTs:\n");
```

```
      printf("      %3.6f seconds were spent on DMAs\n",
         fft_DMA_time);
      printf("      %3.6f seconds were spent in the FFT loop\n",
         fft_fft_time);

      printf("   %3.6f seconds were spent on the CALLS to Channelize\n",
         channel_CALL_time);
      printf("   %3.6f seconds were spent on the MAP for Channelize\n",
         channel_MAP_time);
      printf("   Of the time spent on the MAP for Channelize:\n");
      printf("      %3.6f seconds were spent on DMAs\n",
         channel_DMA_time);
      printf("      %3.6f seconds were spent channelizing\n",
         channel_channel_time);

      printf("  %3.6f seconds were spent on the CALLS to Downconvert\n",
         downconvert_CALL_time);
      printf("   %3.6f seconds were spent on the MAP for Downconvert\n",
         downconvert_MAP_time);
      printf("   Of the time spent on the MAP for Downconvert:\n");
      printf("      %3.6f seconds were spent on DMAs\n",
         downconvert_DMA_time);
      printf("      %3.6f seconds were spent downconverting\n",
         downconvert_downconvert_time);
      printf("Execution time not including calls and data transfers:
%3.6f seconds\n",
         overall_time - fft_CALL_time - fft_DMA_time - channel_CALL_time
-
         channel_DMA_time        -        downconvert_CALL_time        -
downconvert_DMA_time);

   /* PRINT OUTPUT FILE */
      Output=fopen(Output_file, "w");
      if(Output == NULL)
      {
         puts("Error creating output file.");
         return(1);
      }

      for(i=0; i<Np + 1; i++) // Np + 1 for Sx, 5 for MM
      {
         for(j=0; j<2*N + 1; j++)  // 2*N + 1 for Sx, Np*Np for MM
         {
            fprintf(Output, "%6.32f\t", Sx[i][j]);
         }
         fprintf(Output, "\n");
      }
      fclose(Output);

      printf("Results from Cyclostationary FAM algorithm written to:
%s\n", Output_file);

   printf("\nEnd of FAM Program Execution\n");
   return 0;
}
```

C. SRC MAP FILES

1. Channelization MAP File

```
#include <libmap.h>
#include "FAM_const.c"

void channelize (double xx[], double XW_out[], int64_t *t_MAP,
                 int64_t *t_DMA, int64_t *t_channelize, int mapno)
{
   /* Declare Data to Store */
      OBM_BANK_A    (al, double, NN  )
      OBM_BANK_B_2D (X,  double, Np, P)
      OBM_BANK_C_2D (XW, double, Np, P)

   /* Declare Other Variables */
      int64_t i, j, k, nbytes, index, t0, t1, t2, t3;
      double hamming, L = Np/4;

   /* Start MAP timing */
      start_timer();
      read_timer(&t0);

   /* Transfer Data to MAP from CM */
      nbytes = NN * sizeof(double);
      DMA_CPU(CM2OBM, al, MAP_OBM_stripe(1, "A"), xx, 1, nbytes, 0);
      wait_DMA (0);

   /* Take time hac */
      read_timer(&t1);

   /* Channelize: Turn Array into Matrix */
      for (i=0; i<P; i++)
      {
         index = 0;
         for (j=i*L+1; j<=i*L+Np; j++)
         {
            X[index][i] = al[j-1];
            index++;
         }
      }

   /* The following loop was used to generate data for G. Upperman's
      Thesis.  Note: printf statement only works in debug mode. */
      printf("********** THESIS DATA: CHANNELIZATION *******\n");
      for (i=0; i<5; i++)
      {
         printf("\t");
         for (j=0; j<2; j++)
         {
            printf("%2.8f\t", X[i][j]);
         }
         printf("\n");
      }
```

```
   /* Apply Hamming window */
      for (i=0; i<Np; i++)
      {
         hamming = 0.54 - 0.46*cosf(2*pi*i/(Np-1));
         for (j=0; j<P; j++)
         {
            XW[i][j] = hamming*X[i][j];
         }
      } // for i

   /* Get Time Hac */
      read_timer(&t2);

   /* Transfer data from MAP to CM */
      DMA_CPU(OBM2CM, XW, MAP_OBM_stripe(1, "C"), XW_out, 1,
         P * Np * sizeof(double), 0);
      wait_DMA(0);

   /* Get Time Hac and calculate timing */
      read_timer(&t3);
      *t_MAP = t3 - t0;
      *t_DMA = (t3 - t2) + (t1 - t0);
      *t_channelize = t2 - t1;

}
```

2. General FFT MAP File

```
#include <libmap.h>
#include "FAM_const.c"

void cfft_fp32 (int n, int inv, int64_t a_in,   int64_t b_in,
                int64_t c_in,   int64_t d_in,   int sig_valid,
                int64_t e_in,   int64_t f_in,   int twid_valid,
                int starting,   int64_t *a_out, int64_t *b_out,
                int64_t *c_out, int64_t *d_out,
                int *data_valid_out, int *xfrm_addr_out);

void fft_map (float input[], float twiddle[], float output[],
              int n, int frflag, int64_t* t_MAP, int64_t* t_dma,
              int64_t* t_FFT, int map)
{
   int      i, j, starting, inv, loop, npoint;
   int      nbytes, cm_loc, obm_loc;

   int     inidx, outidx, outctr, twididx, sig_len, valid_in, valid_out;
   int      xfrm_addr_out;
   int64_t  a_in, b_in, c_in, d_in, e_in, f_in, a_out, b_out, c_out;
   int64_t  d_out, t0, t1, t2, t3;

   /* input and output MAX_OBM_SIZE */
      OBM_BANK_A (a, int64_t, MAX_OBM_SIZE)
      OBM_BANK_B (b, int64_t, MAX_OBM_SIZE)
      OBM_BANK_C (c, int64_t, MAX_OBM_SIZE)
```

```
      OBM_BANK_D (d, int64_t, MAX_OBM_SIZE)

   /* twiddle table */
      OBM_BANK_E (e, int64_t, MAX_OBM_SIZE)
      OBM_BANK_F (f, int64_t, MAX_OBM_SIZE)

   /* Start MAP timing */
      start_timer();
      read_timer(&t0);

   /* move twiddle table */
      npoint = 1 << n;
      nbytes = npoint*4;
      DMA_CPU(CM2OBM, e, MAP_OBM_stripe(1,"E,F"), twiddle, 1,nbytes, 0);
      wait_DMA(0);

   /* move input data */
      nbytes = npoint*8;
      obm_loc = 0;
      cm_loc  = 0;
      DMA_CPU(CM2OBM, &a[obm_loc], MAP_OBM_stripe(1,"A,B,C,D"),
         &input[cm_loc], 1, nbytes, 0);
      wait_DMA(0);

   /* Take time hac */
      read_timer(&t1);

   /* do fft */
      inv = 0;
      sig_len = npoint/4;
      for (loop=0; loop < frflag; ++loop)
      {
         inidx = 0;
         outidx = 0;
         outctr = 0;
         starting = 1;
#pragma loop noloop_dep
#pragma loop noldst_clsh
         do
         {
            valid_in = ( inidx < sig_len) ? 1 : 0;

            a_in = a[inidx];
            b_in = b[inidx];
            c_in = c[inidx];
            d_in = d[inidx];

            e_in = e[inidx];
            f_in = f[inidx];

            ++inidx;

            cfft_fp32 (n, inv, a_in, b_in, c_in, d_in, valid_in,
                       e_in, f_in, valid_in, starting, &a_out,
                       &b_out, &c_out, &d_out, &valid_out, &outidx);
```

```
            if ( valid_out )
            {
               a[outidx] = a_out;
               b[outidx] = b_out;
               c[outidx] = c_out;
               d[outidx] = d_out;
            }

            cg_accum_add_32_np(1, valid_out, 0, starting, &outctr);
            starting = 0;
         } while ( outctr < sig_len );

         if ( loop == 0 ) {inv = 1;}
      } // for

   /* Take time hac */
      read_timer(&t2);

   /* move output data */
      nbytes = npoint*8;
      obm_loc = 0;
      cm_loc  = 0;

   /* Transfer the data back to the CM */
      DMA_CPU(OBM2CM, &a[obm_loc], MAP_OBM_stripe(1,"A,B,C,D"),
         &output[cm_loc], 1, nbytes, 0);
      wait_DMA(0);

   /* Take time hac and report */
      read_timer(&t3);
      *t_MAP = t3 - t0;
      *t_dma = (t1 - t0) + (t3 - t2);
      *t_FFT = t2 - t1;
}
```

3. Custom FFT MAP File

```
#include <libmap.h>
#include "FAM_const.c"

/* fft(b, a2, a1, n, sgn) -- do an n-point fft of complex vector b and
   return the result in b.  b consists of 2*n FTYPE elements, organized
   as n complex pairs real_1, imag_1, real_2, imag_2, ..., real_n,
   imag_n. the arrays a1 and a2 are used for working storage and each
   has n FTYPE elements.  sgn is 1 for an FFT, and -1 for an IFFT.
   The procedure is taken from the Cormen, Leiserson, and Rivest
   Algorithms text, in the section on efficient FFT implementations.
   This implementation wastes some space; the b array should probably
   be dropped and the initial and  final staging done in-place in
   the "a" arrays.
*/
void fft (double Iin[], double Qin[], double Iot[], double Qot[],
          int  n,  int64_t  *t_MAP,  int64_t  *t_dma,  int64_t  *t_FFT,  int
map)
```

```
{

   /* Declare Arrays in OBM */
      OBM_BANK_A (I,   double, MAX_OBM_SIZE)
      OBM_BANK_B (Q,   double, MAX_OBM_SIZE)
      //OBM_BANK_C (a1, double, 2*Np)
      //OBM_BANK_D (a2, double, 2*Np)
      OBM_BANK_D (temp_I_out, double, MAX_OBM_SIZE)
      OBM_BANK_E (temp_Q_out, double, MAX_OBM_SIZE)

   /* Declare Other Variables */
      int64_t nbytes, i, j, k, k2, s, log2n, time0, time1, time2, time3;
      int m, o;
      unsigned int ii, p, q;
      double wm1, wm2, w1, w2, t0, t1, t2, u1, u2;

      float a1[(int)Np], a2[(int)Np];

   /* Get Initial Time Hac */
      read_timer(&time0);

   /* Transfer Data */
      nbytes = n * sizeof(double);

      DMA_CPU(CM2OBM, I,  MAP_OBM_stripe(1, "A"), Iin,  1, nbytes, 0);
      wait_DMA(0);

      DMA_CPU(CM2OBM, Q,  MAP_OBM_stripe(1, "B"), Qin,  1, nbytes, 0);
      wait_DMA(0);

   /* Take time hac */
      read_timer(&time1);

   /* Determine Log Base 2 */
      for (i=8*sizeof(int)-1; i>=0 && ((1<<i) & n)==0; i--);
      log2n = i;

   /* reorder input and split input into real and complex parts */
      for (i=0; i<n; i++)
      {
         /* reverse bits 0 thru k-1 in the integer "a" */
            for (ii=o=0, p = 1, q = 1<<(log2n-1);
               ii<log2n;
               ii++, p <<= 1, q >>= 1 ) if (i & q) o = o | p;

         j = (int)o;
         a1[j] = I[i];
         a2[j] = Q[i];
      }

   /* loop on FFT stages */
      for (s=1; s<=log2n; s++)
      {
         m = 1<<s;  /* m = 2^s */
         t0  = 2*pi/m;
```

```
         wm1 = cosf(t0);   /* wm = exp(q*2*pi*i/m); */
         wm2 = sinf(t0);

         w1 = 1.0;
         w2 = 0.0;

         for (j=0; j<m/2; j++)
         {
            for (k=j; k<n; k+=m)
            {
               /* t = w*a[k+m/2]; */
               k2 = k+ m/2;

               u1 = a1[k2];
               u2 = a2[k2];

               t1  = w1 * u1 - w2 * u2;
               t2 =  w1 * u2 + w2 * u1;

               u1 = a1[k];
               u2 = a2[k];

               a1[k] = u1 + t1;
               a2[k] = u2 + t2;

               a1[k2] = u1 - t1;
               a2[k2] = u2 - t2;
            } // for k

         /* w = w * wm; */
         t1 = w1 * wm1  -  w2 * wm2 ;
         w2 = w1 * wm2  +  w2 * wm1 ;
         w1 = t1;
      } // for j
   } // for s

   /* flip the final stage */
      temp_I_out[0] = a1[0];
      temp_I_out[(int)n/2] = a1[(int)n/2];
      temp_Q_out[0] = a2[0];
      temp_Q_out[(int)n/2] = a2[(int)n/2];

#pragma src parallel sections
   {
#pragma src section
      {
         for (i=1; i<n/2; i++)  {temp_I_out[i] = a1[n-i];}
         for (i=1; i<n/2; i++)  {temp_I_out[n-i] = a1[i];}
      }
#pragma src section
      {
         for (j=1; j<n/2; j++)  {temp_Q_out[j] = a2[n-j];}
         for (j=1; j<n/2; j++)  {temp_Q_out[n-j] = a2[j];}
      }
   }
```

```
   /* Take time hac */
      read_timer(&time2);

   /* Transfer data back */
      DMA_CPU(OBM2CM, temp_I_out, MAP_OBM_stripe(1, "D"), Iot, 1,
         nbytes, 0);
      wait_DMA(0);

      DMA_CPU(OBM2CM, temp_Q_out, MAP_OBM_stripe(1, "E"), Qot, 1,
         nbytes, 0);
      wait_DMA(0);

   /* Take time hac and report */
      read_timer(&time3);
      *t_MAP = time3 - time0;
      *t_dma = (time1 - time0) + (time3 - time2);
      *t_FFT = time2 - time1;
}
```

4. Downconversion MAP File

```
#include <libmap.h>
#include "FAM_const.c"

void downconvert (double Iin[], double Qin[],
                  double twiddleI[], double twiddleQ[], double Iout[],
                  double Qout[], int64_t *t_MAP, int64_t *t_DMA,
                  int64_t *t_downconvert, int mapno)
{
   /* Get Space in OBM Banks */
      OBM_BANK_A_2D (IonA, double, Np, P)
      OBM_BANK_B_2D (QonB, double, Np, P)

      OBM_BANK_E_2D (twdlI, double, Np, P)
      OBM_BANK_F_2D (twdlQ, double, Np, P)

      OBM_BANK_C_2D (IXM, double, P,  Np*Np)
      OBM_BANK_D_2D (QXM, double, P,  Np*Np)

   /* Declare Other Variables */
      int64_t i, j, k, nbytes, t0, t1, t2, t3;
      double L = Np/4;
      float Ii, Qi, Ij, Qj;  //temporary I and Q scratch variables
      double IXF1[(int)Np][(int)P], QXF1[(int)Np][(int)P];
      double  IXE[(int)P][(int)Np],  QXE[(int)P][(int)Np];

   /* Start MAP timing */
      start_timer();
      read_timer(&t0);

   /* Transfer data over to MAP */
      nbytes = Np * P * sizeof(double);
```

```
      DMA_CPU(CM2OBM, IonA, MAP_OBM_stripe(1, "A"), Iin, 1, nbytes, 0);
      wait_DMA (0);

      DMA_CPU(CM2OBM, QonB, MAP_OBM_stripe(1, "B"), Qin, 1, nbytes, 0);
      wait_DMA (0);

      DMA_CPU(CM2OBM, twdlI, MAP_OBM_stripe(1, "E"), twiddleI, 1,
         nbytes, 0);
      wait_DMA (0);

      DMA_CPU(CM2OBM, twdlQ, MAP_OBM_stripe(1, "F"), twiddleQ, 1,
         nbytes, 0);
      wait_DMA (0);

   /* Get initial time hac */
      read_timer(&t1);

   /* Implement FFT shift: End Result swaps the top and bottom halves:*/
      for(i=0; i<(Np/2); i++)
      {
         for(j=0; j<P; j++)
         {
            /* Bottom real half becomes top real half */
               IXF1[i][j] = IonA[i+(int)(Np/2)][j];
            /* Top real half becomes bottom real half */
               IXF1[i+(int)(Np/2)][j] = IonA[i][j];

            /* Bottom imag half becomes top imag half */
               QXF1[i][j] = QonB[i+(int)(Np/2)][j];
            /* Top imag half becomes bottom imag half */
               QXF1[i+(int)(Np/2)][j] = QonB[i][j];
         } // for j
      } // for i

   /* The following nested loop was used to generate data for G.
      Upperman's Thesis.  Note: printf only works in debug mode */
      printf("********* THESIS DATA: FFT 1 AND SHIFT *****\n");
      for (i=0; i<5; i++)
      {
         printf("\t%2.8f+i*%2.8f\n", IXF1[i][0], QXF1[i][0]);
      }

   /* Downconversion - the short sliding FFT's results are shifted
      to baseband to obtain decimated complex demodulate sequences.
      The transpose of the matrix is taken at the same time. */
      for(i=0; i<Np; i++)
      {
         for(j=0; j<P; j++)
         {
            Ii = twdlI[i][j];
            Qi = twdlQ[i][j];
            IXE[j][i] = (IXF1[i][j] * Ii) - (QXF1[i][j] * Qi);
            QXE[j][i] = (IXF1[i][j] * Qi) + (QXF1[i][j] * Ii);
         }
```

```
      }

  /* Multiplication - the product sequences between each one of the
     complex demodulates and the complex conjugate of the others
     are formed.  This forms the area in the bi-frequency plane. */
     for (i=0; i<Np; i++)
     {
        for (j=0; j<Np; j++)
        {
           for (k=0; k<P; k++)
           {
              Ii = IXE[k][i];
              Qi = QXE[k][i];
              Ij = IXE[k][j];
              Qj = QXE[k][j];
              IXM[k][i*(int)Np+j] =  (Ii * Ij) + (Qi * Qj);
              QXM[k][i*(int)Np+j] = -(Ii * Qj) + (Qi * Ij);
           } // for k
        } // for j
     } // for i

   /* Get time hac */
      read_timer(&t2);

   /* Transfer results back to the CM */
      DMA_CPU(OBM2CM, IXM, MAP_OBM_stripe(1, "C"), Iout, 1,
         Np * nbytes, 0);
      wait_DMA(0);

      DMA_CPU(OBM2CM, QXM, MAP_OBM_stripe(1, "D"), Qout, 1,
         Np * nbytes, 0);
      wait_DMA(0);

   /* Get last time hac and report */
      read_timer(&t3);
      *t_MAP = t3 - t0;
      *t_DMA = (t3 - t2) + (t1 - t0);
      *t_downconvert = t2 - t1;

}
```

D. SRC MAKEFILE

```
FILES       = FAM.c

MAP_E_FILES = channelize.mc \
              fft.mc \
              downconvert.mc

BIN         = FAM

SRC_FFT_LIB = /opt/SRCCI2.2/fft_lib/
SRC_VERSION = comp
SRC_TARGET  = map_e
```

```
# ----------------------------------
# Multi chip info provided here
# (Leave commented out if not used)
# ----------------------------------

#----------------------------------
# User defined directory of code routines
# that are to be inlined
#----------------------------------
MAPTARGET   = map_e

# ----------------------------------
# User defined macros info supplied here
# (Leave commented out if not used)
# ----------------------------------

# ----------------------------------
# Floating point macros selection
# ----------------------------------

# ----------------------------------
# User supplied MCC and MFTN flags
# ----------------------------------
MCCFLAGS    = -log
MFTNFLAGS   = -log

# ----------------------------------
# User supplied flags for C & Fortran compilers
# ----------------------------------
CC          = icc       # icc   for Intel cc for Gnu
FC          = ifort     # ifort for Intel f77 for Gnu
LD          = icc       # for C codes

USER_MACROLIBS = $(SRC_FFT_LIB)
LDFLAGS = -lSDL

# ----------------------------------
# VCS simulation settings
# (Set as needed, otherwise just leave commented out)
# ----------------------------------

# ----------------------------------
# No modifications are required below
# ----------------------------------
MAKIN   ?= $(MC_ROOT)/opt/srcci/comp/lib/AppRules.make
include $(MAKIN)

mydebug: debug

myhw: hw

myclean: clobber
      rm -rf *~
```

E. C CODE TO GENERATE CONSTANT INCLUDE FILE

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>

FILE *FAM_const;

main()
{
   /* DECLARE VARIABLES AND CONSTANTS */
      /* declare file names and path */
         char const_file[] = "FAM_const.c";
      /* declare sample frequency, frequency resolution, and M */
         int fs, df, M;
      /* declare FAM-spcific varibles */
         double dalpha, Np, L, P, N, NN;

   /* Get data from user */
      printf("What is the sampling frequency (fs)(Hz)? ");
      scanf("%d", &fs);
      printf("What is the frequency resolution desired (df)(Hz)? ");
      scanf("%d", &df);
      printf("What is M? ");
      scanf("%d", &M);

   /* Calculate Values */
      dalpha = df/M;
      Np = pow(2.0, ceil(log10(fs/df)/log10(2)) );
      L = Np/4;
      P = pow(2.0, ceil(log10(fs/dalpha/L)/log10(2)) );
      N = P*L;
      NN = (P-1)*L+Np;

   /* PRINT CONSTANT FILE */
      FAM_const=fopen("FAM_const.c", "w");
      if(FAM_const == NULL)
      {
         printf("Error creating FAM constant file.");
         return(1);
      }
      fprintf(FAM_const, "#define N %i\n", (int)N);
      fprintf(FAM_const, "#define NN %i\n", (int)NN);
      fprintf(FAM_const, "#define Np %i\n", (int)Np);
      fprintf(FAM_const, "#define P %i\n", (int)P);
      fprintf(FAM_const, "\n#define pi 3.141592653589793\n");

      fclose(FAM_const);

   /* END FUNCTION */
      printf("\nN: %i, NN: %i, Np: %i, P: %i\n",
         (int)N, (int)NN, (int)Np, (int)P);
      printf("Constant   File   written   for   Cyclostationary   FAM
Analysis.\n\n");
```

```
      return 0;
}
```

F. MATLAB CODE TO GENERATE OUTPUT PLOTS

```
% Load Output from C Program
   filename = 'FAM_result.txt';

C_FAM = load(['Y:\thesis\with_fft\SRC_filesV1\', filename], ...
   'ascii'); temp = 'SRC';

if (size(C_FAM) == [65 257])
   % The following was the data used for G. Upperman's Thesis
   disp('*********** THESIS DATA: OUTPUT ***********')
   disp(C_FAM(31:35, 8:9))
end

N = 128; fs = 7000; Np = 64;  % for Frank and FMCW
%N = 256; fs = 15057; Np = 128; % for Costas and FSK/PSK C
alpha0 = -fs:fs/N:fs;
f0 = -fs/2:fs/Np:fs/2;

figure
contour (alpha0, f0, C_FAM); grid;
xlabel('Cycle frequency (Hz)'); ylabel('Frequency (Hz)');
title (['Time Smoothing SCD from ', temp, ' ', filename, ', df = ',
   int2str(128),', N = ', int2str(N)]);

axis([-250 250 800 1200]), title('')  %for Frank Signal used in thesis
%axis([-800 800 600 1600]), title('')  % for FMCW
%axis([1500 14500 -3500 3500]), title('') % for Costas and FSK/PSK C
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] K. M. Stoffell, “Implementation of a Quadrature Mirror Filter Bank on an SRC Reconfigurable Computer for Real-Time Signal Processing,” M.S. thesis, Naval Postgraduate School, Monterey, CA, 2006.

[2] T. L. O. Upperman, “ELINT Signal Processing Using Choi-Williams Distribution on Reconfigurable Computers for Detection and Classification of LPI Emitters,” M.S. thesis, Naval Postgraduate School, Monterey, CA, 2008.

[3] P. E. Pace, *Detecting and Classifying Low Probability of Intercept Radar*, Boston, MA: Artech House Inc., 2004.

[4] K. R. Macklin, “Benchmarking and Analysis of the SRC-6E Reconfigurable Computing System,” M.S. thesis, Naval Postgraduate School, Monterey, CA, 2003.

[5] K. R. Macklin, “Suitability of the SRC-6E Reconfigurable Computing System for Generating False Radar Images,” M.S. thesis, Naval Postgraduate School, Monterey, CA, 2004.

[6] T. G. Guthrie, “Design, Implementation, and Testing of a Software Interface Between the AN/SPS-65(V)1 Radar and the SRC-6E Reconfigurable Computer,” M.S. thesis, Naval Postgraduate School, Monterey, CA, 2005.

[7] T. L. King, “Hardware Interface to Connect an AN/SPS-65 Radar to an SRC-6E Reconfigurable Computer,” M.S. thesis, Naval Postgraduate School, Monterey, CA, 2005.

[8] D. A. Brown, “ELINT Signal Processing on Reconfigurable Computers for Detection and Classification of LPI Emitters,” M.S. thesis, Naval Postgraduate School, Monterey, CA, 2006.

[9] S. P. Bailey, “Neural Network Design on the SRC-6 Reconfigurable Computer,” M.S. thesis, Naval Postgraduate School, Monterey, CA 2006.

[10] W. A. Gardner and C. M. Spooner, “Signal interception: performance advantages of cyclic-feature detectors,” *IEEE Transactions on Communications*, vol. 40, no. 1, pp. 149-159, January 1992.

[11] SRC Computers, Inc., “SRC Carte training course,” presented by David Caliga at a private training session, Colorado Springs, CO, July 2007.

[12] A. M. Dewey, *Analysis and Design of Digital Systems with VHDL*. Boston, MA: PWS Publishing Company, 1997, pp. 146-149.

[13] “SRC Carte C Programming Environment v2.2 Guide,” SRC-007-18, SRC Computers, Inc., Colorado Springs, CO, August 21, 2006.

[14] W. A. Brown, III and H. H. Loomis, Jr., “Digital implementations of spectral correlation analyzers,” *IEEE Transactions on Signal Processing*, vol. 41, no. 2, pp. 703-720, February 1993.

[15] Wikimedia Foundation, Inc., “Window Function,” February 8, 2008. [Online]. Available: Wikipedia, http://wikipedia.org [Accessed: February 26, 2008].

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 2nd ed. Cambridge, MA: The MIT Press, 2002.

[17] J. Squire, “Jon Squire”, April, 2007. [Online]. Available: http://www.csee.umbc.edu/~squire/cs455_l18.html [Accessed: 4 March, 2008].

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Chairman, Code EC
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

4. Douglas J. Fouts
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

5. Phillip E. Pace
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

6. Peter K. Burke
   771 Test Squadron
   Edwards, California

7. David Caliga
   SRC Computers, Inc.
   Colorado Springs, Colorado

8. Jon Huppenthal
   SRC Computers, Inc.
   Colorado Springs, Colorado

9. Alan Hunsberger
   National Security Agency
   Ft. Mead, Maryland

10. Ted Roberts
    Naval Research Laboratory
    Code 5720
    Washington, D. C.

11. Anthony Tse
Naval Research Laboratory
Code 5720
Washington, D. C.

12. Alfred Di Mattesa
Naval Research Laboratory
Code 5701
Washington, D. C.

13. Peter Craig
Office of Naval Research
Code 312
Arlington, Virginia

14. Jon Squire
Department of Computer Science and Electrical Engineering
Univeristy of Maryland Baltimore County
Baltimore, Maryland